

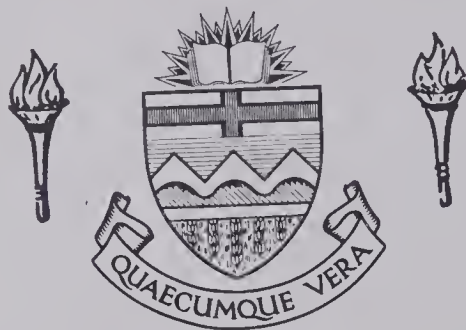
For Reference

NOT TO BE TAKEN FROM THIS ROOM

For Reference

NOT TO BE TAKEN FROM THIS ROOM

Ex libris UNIVERSITATIS ALBERTAENSIS




Regulations Regarding Theses and Dissertations

[illegible]

THE UNIVERSITY OF ALBERTA

INPUT AND OUTPUT LANGUAGES

by

 J. David Thomson

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES
IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE DEGREE
OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTING SCIENCE

EDMONTON, ALBERTA

October, 1968

1hca/3
1969
135

UNIVERSITY OF ALBERTA

FACULTY OF GRADUATE STUDIES

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies for acceptance, a thesis entitled INPUT AND OUTPUT LANGUAGES submitted by J. David Thomson in partial fulfilment of the requirements for the degree of Master of Science.

ABSTRACT

This thesis reviews and illustrates the most common codes for representation of data items in a computer memory or other storage device, and the representation of data structures in memory. The operations of input and output, and the conversions which may be required, are reviewed.

Linguistic terms, particularly syntax and semantics, are applied to data representations, to data structures, and to format statements. Syntax-directed analysis of format statements is demonstrated by algorithms coded in APL. The applicability of these algorithms to the standardization of format languages is discussed.

ACKNOWLEDGEMENTS

I wish to express my appreciation to Professor W.S. Adams for his guidance in the preparation of this thesis, and to the Department of Computing Science for the computing facilities and financial assistance which have made this research possible.

TABLE OF CONTENTS

	Page
CHAPTER I - INTRODUCTION	1
CHAPTER II - DATA IN COMPUTER MEMORY	
2.1 Physical Representations of Data	5
2.2 Input and Output	8
2.3 Data Structures	9
2.4 DATA - A Simple Programming Language	14
CHAPTER III - INPUT AND OUTPUT	
3.1 Record-Oriented and Item-Oriented Input and Output	27
3.2 Item-Oriented I/O of DATA, Fortran and PL/1	28
3.3 Record-Oriented I/O of DATA, Cobol and PL/1	32
3.4 I/O Conversions	35
CHAPTER IV - SYNTAX-DIRECTED ANALYSIS OF FORMAT LANGUAGES	
4.1 Introduction	39
4.2 A Parsing Algorithm	40
4.3 An Experimental Universal Format Language	49
4.4 Evaluation and Possible Future Extensions	59
CHAPTER V - CONCLUSION	
5.1 Languages	64
5.2 N-Dimensional Languages	66
5.3 Input and Output	66
5.4 A Proposal for the Standardization of Format Languages	67
5.5 DATA as a Teaching Aid	70
BIBLIOGRAPHY	72

	Page
APPENDIX A - NUMERIC DATA IN MEMORY	
A.1 Numbers and their Representations	75
A.2 Fixed-Point Representations	79
A.3 Floating-Point Representations	82
APPENDIX B - DATA PROGRAMMER'S GUIDE	
Chapter B1 - Introduction	86
B.1.1 Definitions of Terms	86
B.1.2 Memory Organization	88
B.1.3 Data Representations	88
B.1.4 Reserved Words	90
Chapter B2 - Declarative Statements	92
Chapter B3 - Assignment Statements	95
Chapter B4 - Input and Output Statements	100
Chapter B5 - Control Statements	105
Appendix to DATA Programmer's Guide	108
AB.1 The DATA Picture Language	108
AB.2 Syntax of the Picture Language	108
AB.3 Semantics of the Picture Language	108
AB.3.1 Usage	108
AB.3.2 Digit Positions	112
AB.3.3 Insertions	113
AB.3.4 Qualifiers	113
AB.3.5 Signs	113
APPENDIX C - TRANSITION DIAGRAMS AND APL FUNCTIONS	
C.1 Format Languages	115
C.2 Listing of Functions	124
C.2.1 The DATA Model	124
C.2.2 The Universal Format Language (UFL) System	137

LIST OF TABLES

	Page
B.1	109
B.2	111

LIST OF FIGURES

	Page
2.1	18
2.2	19
2.3	21
2.4	23
3.1	30
3.2	33
4.1	41
4.2	42
4.3	43
4.4	47
4.5	50
4.6	51
4.7	57
4.8	61
C.1	116
C.2	118
C.3	119
C.4	122
C.5	123

CHAPTER I

INTRODUCTION

In our everyday experience, information is transmitted in many different forms, including speech sounds, letters and numerals on paper, and holes in punched cards. Where the recipient of the information is a computer, however, a translation is required into one of the comparatively few forms which the computer can process. Because of physical properties of the storage media, binary representations are used exclusively within the computer and on external storage devices. However, the numeric information directed to the computer and the numeric results received from it will almost invariably be represented in decimal for the convenience of humans. These are the two basic problems of input and output: the many-to-one transformations between the diverse preferences of humans and the restricted repertoire of the machine, and the necessity for decimal-to-binary and binary-to-decimal conversions.

The obvious method of converting a decimal representation to binary is by treating it as a vector of decimal digits, encoding each digit as a binary integer. Alternatively, the number as a whole may be represented in binary, the decimal digits being discarded. Control of the transformations between internal and external representations is achieved by

use of a "format language" which is capable of describing concisely the most commonly-used external representations. A language is simply a set of rules which assigns meaning to a string of symbols. In this sense, the Arabic numeral system is a language, as is each system of representing information in a computer memory. The linguistic concepts of syntax and semantics are applicable to these languages. Syntax refers to the structure, that is, the permissible arrangements of symbols. Semantics refers to the meaning assigned to a group of symbols, and the set of transformations which are defined on it.

The hierarchy of units of information undergoing any input or output operation consists of fields, records, and blocks. The block, the most inclusive of the three units, is the smallest unit which can be physically transmitted between the computer and an external storage device. The field, the least inclusive, is the smallest unit to which the program has access. A record consists of one or more fields, and a block contains one or more records. Any programming language statement which initiates an input or output operation has access to only one level of this hierarchy. A statement which can initiate transmission of a field will be called an item-oriented statement. One which can initiate transmission of a complete record only will be called record-oriented. There are also assembly-language

instructions which could be called "block-oriented", but they are beyond the scope of this thesis.

A model of a computer with storage devices, illustrating the processes of input and output, is described in Chapter II. This model is designed to represent the I/O operations of a typical computer, while avoiding the technical restrictions imposed by an actual implementation. For example, the memory organization of the model is similar to that of the IBM 360 in that both "field-oriented" and "word-oriented" accessing are possible. However, the 360 requires that the programmer recognize word boundaries, while the model does not.

A second model, intended to work in conjunction with the first, uses syntax-directed analysis to translate Fortran and PL/I format statements into a common notation which can be executed to perform data conversions. This notation (or an extension of it) is proposed as a basis for the standardization of format languages.

The algorithms which make up these models are coded in APL, a conversational language based on Iverson's language (Iverson (1962)) and described fully in Falkoff and Iverson (1967) and Pakin (1970). Where APL expressions are used in the text of the thesis, the symbol "t" preceding a conditional expression implies "truth", a boolean value of 1 (as in Pakin (1970)). Thus the statement:

The first part of the document discusses the importance of maintaining accurate records of all transactions. It emphasizes that proper record-keeping is essential for the transparency and accountability of the organization. The document outlines the various methods used to collect and analyze data, ensuring that the information is reliable and up-to-date. It also mentions the role of technology in streamlining these processes and reducing the risk of errors.

The second part of the document focuses on the financial aspects of the organization. It provides a detailed overview of the budget, including the projected income and expenses for the upcoming year. The document highlights the need for careful financial management to ensure that the organization remains solvent and able to meet its obligations. It also discusses the importance of regular financial reviews and the role of the board of directors in overseeing the financial health of the organization.

The third part of the document addresses the operational challenges faced by the organization. It identifies the key areas where improvements are needed, such as enhancing the efficiency of the supply chain and improving the quality of customer service. The document proposes several strategies to address these challenges, including the implementation of new technologies and the hiring of additional staff. It also mentions the importance of ongoing training and development for the existing workforce.

The fourth part of the document discusses the organization's commitment to social responsibility and environmental sustainability. It outlines the various initiatives that the organization has implemented to reduce its carbon footprint and support the local community. The document emphasizes the importance of these efforts in building a positive reputation for the organization and ensuring its long-term success.

The final part of the document provides a summary of the key findings and recommendations. It reiterates the importance of maintaining accurate records, managing finances carefully, addressing operational challenges, and committing to social responsibility. The document concludes by expressing the organization's confidence in its ability to achieve its goals and its commitment to the stakeholders who support it.

$$t \wedge 0 \ 1 = F \geq 1, \ R \star^{-1}$$

(from the discussion of floating-point representations in Appendix A) states an identity which holds for each normalized floating-point representation.

Where indices or related operations appear, l-origin indexing is used.

CHAPTER II

DATA IN COMPUTER MEMORY

2.1 Physical Representations of Data

A number is a quantity or a measure, the result obtained by counting or measuring or by performing a computation upon such results. A representation of a number consists of a string of symbols, such as numerals printed on a page, holes in a card or magnetic or electrical states of a circuit, to each of which one or more meanings have been assigned. Any system of assigning meanings to symbols is, in effect, a written language to which the terminology and methods of linguistics are applicable.

The alphabet of a language is the set of all symbols which may appear in a string. The syntax refers to the possible linear arrangements of these symbols. A rule of syntax states a permissible or prohibited relation between symbols. Semantics defines the relationship between a symbol and the set of meanings attributed to it (McIntosh (1968)). In other words, syntax refers to the structure of a string and semantics refers to its meaning. Symbols which may be used singly or in groups to represent numbers are called digits. Rules of syntax define permissible arrangements of digits, while rules of semantics may be used to identify the number which is represented by any such arrangement.

Any representation of numeric or non-numeric data, whether

inside a computer or elsewhere, is essentially a method of identifying one of a set of possible values. Appendix A describes the most commonly-used "languages" for the representation of numbers in a computer. However, it is clear that no string of symbols is inherently numeric. A system for representing the integers from 1 to N, for example, is equally suitable for the representation of any N-symbol alphabet. All that is required is the establishment of a one-to-one correspondence (called a code) between that alphabet and the set of integer representations. (Where the alphabet is ordered, this correspondence is equivalent to the operation of subscripting a vector of symbols.)

In a digital computer, data is represented by the discrete states of physical devices. Practically all types of these devices operate in a bistable manner. That is, they have the properties:

1. Each device may assume either of two distinguishable states.
2. Each may be set in either state, and will retain that state indefinitely until changed.

Such a device, which we shall call a storage cell, is the fundamental storage element of the computer. The smallest grouping of storage cells in the memory of a computer is called a byte (a word coined by IBM. Buchholz (1962)). A group of bytes which is transmitted to or from memory in a

single memory cycle is called a word. The lengths of the byte and the word are thus structural properties of the memory. The term "character" often implies a graphic symbol, such as a letter or numeral, which can be encoded in a byte. We shall refer to the contents of a byte as a character whether or not a corresponding graphic symbol exists. Where such a symbol does exist, it is likely to be represented by different characters for different applications. For example, the alphanumeric and binary fixed-point representations of the symbol "5" are likely to be different. A named collection of data is called a data set. Where it resides upon an external storage device such as a magnetic tape or disk, we shall call it an external data set. A data item is any single number or non-numeric value. A field is a group of characters (or bits), residing either in memory or in an external data set, which represents a single data item. It is the smallest unit of data in memory which can be accessed by the program.

The machine designer has a choice between assigning an address to each byte or to each word. The choice is influenced by the types of problem (i.e., the types of data) which the machine is expected to encounter most frequently. For computational problems, where each representation of a number is assigned to a single word (or to two or more consecutive words for greater precision), a word-addressable memory is more appropriate. Where alphanumeric data is to

be processed, however, the requirement that each field length be some multiple of the word length will prove to be unduly restrictive unless the word length is one byte, i.e., unless memory is byte-addressable. A machine which is intended to process both numeric and alphanumeric data may have its memory organized in a combination of these modes. Alternatively, the second mode may be provided by "software". For example, the Cobol compiler for the IBM 7040 provides variable field length accessing of a word-addressable memory.

2.2 Input and Output

The operations of transmitting data from an external data set to memory, and vice versa, are referred to as input and output respectively. The abbreviation I/O refers to input or output or both. The data representations described in Appendix A may be used both in memory and in external storage devices. The physical unit of data transmission, that is, the smallest collection of data items which can be physically passed between memory and an external data set, is called a block. Briefly, the output of a block proceeds as follows:

A number of fields are concatenated (in the order specified by the program), to form a unit called a record, which is transferred to the area of memory called the output buffer. Subsequent records are constructed until the number required to form a block have been transferred to the buffer.

This block is then transmitted as a unit to the external storage device.

The process of input consists of the reverse sequence of events. A block from the storage device is transmitted into the input buffer, where its component records become accessible. The program obtains access to one record at a time, subdividing it into fields for processing. The length of a block, typically several hundred or a few thousand bytes, depends in part upon physical characteristics of the storage device. A record is the only unit of data which the program can receive from, or pass to, the buffer. It is a logical unit of data, that is, a grouping of data on the basis of its content. The length (or size) of a record is determined by the logic of the program, whereas the length of a block depends upon physical requirements. Consequently, the record is sometimes called a logical record, and the block a physical record.

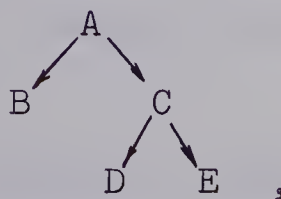
2.3 Data Structures

A record is an example of what we shall call a data structure, since it is a grouping of fields which can be referenced as a unit. Another example is the array. A data structure may appear to the programmer (regardless of the programming language he uses) as one or more of the following:

1. A vector of characters, formed by the concatenation of a number of fields. This characterization is close to the physical reality, and is a useful way of visualizing a record undergoing I/O.
2. A hierarchical arrangement of identifiers, each of which accesses a particular field. As an illustration, consider the following APL functions:

$\nabla R \leftarrow A$	$\nabla R \leftarrow C$
$R \leftarrow B, C$	$R \leftarrow D, E$
∇	∇

Where B , D and E are all scalars or vectors of the same type (i.e., all numeric or all non-numeric), any reference to A , B , C , D or E (provided that a function name does not appear to the left of a specification arrow) will identify some subset of the current value of the vector B , D , E . These five identifiers will behave as if they were members of the hierarchy



since any change in the value of B , D or E will

immediately affect the result returned by an identifier above it on the hierarchy.

3. An algorithm for the selection of any desired group of one or more data items by means of identifiers and indices. As an illustration:

In both Fortran and PL/1, the notation used to refer to an element of an array of N dimensions is identical to that used to call a function of N integer arguments (called a "procedure" in PL/1 and a "function subprogram" in Fortran). The statement "X = TABLE (3,4)" in either language is assumed to be a function call unless TABLE has been declared as a matrix. Because of this notational similarity, array identifiers and function names are interchangeable (except when they appear on the left hand side of an assignment statement. The statement "TABLE (3,4) = X" is valid only where TABLE is a matrix.). A programmer may use this interchangeability to test validity of subscripts, to allow negative subscripts where the language itself does not, to conserve memory when storing a triangular matrix, etc.

The APL function Λ , described in (2) above, serves as a second example. Any APL expression may be indexed. Thus where

$$\vdash 6 = \rho B$$

$$\vdash 4 = \rho D ,$$

it follows that

$$\vdash A[4] = B[4]$$

$$\vdash A[8] = D[2]$$

$$\vdash \wedge/E = ((\rho A)\omega 10)/A .$$

The term syntax was used in Chapter I to refer to structural properties of data representations. "Inherent structural properties" (Wegner (1968)) of data structures will also be called syntactic properties. As an illustration of the similarity between a single representation and a structure, note that the BCD representation of a number consists of the binary fixed point representations of the digits of its decimal representations. Thus the decimal representation is encoded as if it were a structure (a vector) of digits. An array is frequently a representation of some other entity. For example, a matrix may represent

1. A linear transformation, possessing a determinant, an inverse, and eigenvalues.

2. A function of integer arguments, as in the IBM 1620 computer, where decimal arithmetic is performed by a table-lookup procedure. The addition operation consists of a matrix, which we shall call *SUM*, such that

$$\dagger \text{ SUM}[I;J] = I + J .$$

The matrices of these two examples differ in what we shall call their semantic properties, that is, the set of operations which can be performed upon them. For example, it would be meaningless to attempt to find eigenvalues of *SUM*. By the semantic properties of a data structure, we shall mean the set of transformations which it can "initiate or undergo" (Wegner (1968)). (The data structures mentioned so far cannot "initiate" any transformation. However, format statements, introduced in Chapter III, can.)

The semantic properties of any data representation include the semantic properties of the class of representations to which it belongs, plus the "meaning" of the particular representation. The semantic properties of a class of numeric representations, for example, include the capacity to undergo arithmetic operations, rounding, radix conversion, etc. Also included is the set of rules by which any member of that class may be interpreted, that is, by which its meaning may be found.

2.4 DATA - A Simple Programming Language

The essential steps of input and output operations on any computer were described in Section 2.2. The degree of control which the programmer can exercise over the process depends largely upon the programming language used. The following chapter will discuss the statements used in several programming languages to specify I/O operations and data conversions. Such a discussion can be expressed most clearly by referring to a particular example, but an example which does not introduce unnecessary restrictions is difficult to find. An advantage of the IBM 360/67 for the purpose is the fact that it combines the "word-oriented" and "field-oriented" modes of memory organization, and can therefore demonstrate the I/O processes of either. However, the requirement that the programmer be familiar with the word-boundary alignment restrictions tends to outweigh that advantage. The example which will be used is a model called DATA, consisting of a set of APL functions and arrays. Since both the memory and the external storage device of this model are represented by matrices, there are no physical restrictions upon either. Its conversational mode gives it the additional advantage of quick response to a user's input. The DATA Language consists of statements which are intended to resemble the corresponding statements of actual compiler languages. A binary matrix called *CORE* contains representations of data similar to those

used in the memories of actual computers. Since each row of *CORE* represents a byte, the row index serves as a byte address.

Some features of DATA are:

1. 6-bit byte - Results in a character set (the vector *DATASEQ*) of 64 characters. 5 bits would be insufficient to encode 26 alphabetic characters plus 10 digits, while 7 or more would make it impossible to assign a unique graphic symbol to each character without the inconvenience of overstriking.

2. 4 internal representations:

Fixed-point binary	1 to 6 bytes
Floating-point binary	4 or 7 bytes
BCD numeric	Any length, not exceeding one record.
Alphanumeric	

The IBM 360 "packed decimal representation, requiring an 8-bit byte, is not included. Since bytes are addressed in *CORE* and no word boundaries exist, the lengths of long and short floating-point representations can be chosen independently. One need not be a multiple of the other.

Arrays can be defined, subject to several restrictions.

3. Statements resemble those of Cobol and PL/1 -

For example:

DATA : 'A' MOVETO 'B'

('A' OF 2) MOVETO 'B' OF 1

COBOL : MOVE A TO B.

MOVE A OF IN-REC TO B OF OUT-REC.

DATA : 3 WRITEFROM 2

PL/1 : WRITE(FILE-3) FROM(REC-2);

Statements of the DATA language fall into five classes:

Declarative - to define fields and assign an
identifier and a picture to each.

Assignment - to store or retrieve the contents of
a field.

Input/output - to cause data to be transmitted between
memory and the external data set
(*DATASET*).

Control - to create, print or rewind a data set.

APL - provides the operators for arithmetic,
function definition, branching, etc.

For more details, see the DATA Programmer's Guide Appendix B.

CORE consists of one or more records, which the programmer can subdivide into fields, assigning an identifier and a type declaration to each field. A type declaration indicates which representation is required, and the relationship of this field to other fields (e.g., their relative positions in a record or an array).

Every data structure is stored in memory in the form of a vector of characters. To relate the hierarchical form of the structure seen by the programmer to this linear arrangement in storage, we shall introduce some terminology from graph theory.

The most general figure which we shall require is the directed graph, which consists of a vector of nodes, and an arbitrary set of undirectional associations between pairs of its components. A directed graph which contains no circuits will have one or more initial nodes and one or more final nodes (see Figure 2.1). With the added constraint that at most one branch may enter any node, the graph becomes a tree, with initial and final nodes called roots and leaves, respectively. This definition ensures that any path from a root to a leaf will be unique. Any array M may be represented as a $(\rho M)[1]$ - tuply rooted tree (i.e., a tree of $(\rho M)[1]$ roots), as in Figure 2.2. (Iverson (1962)).

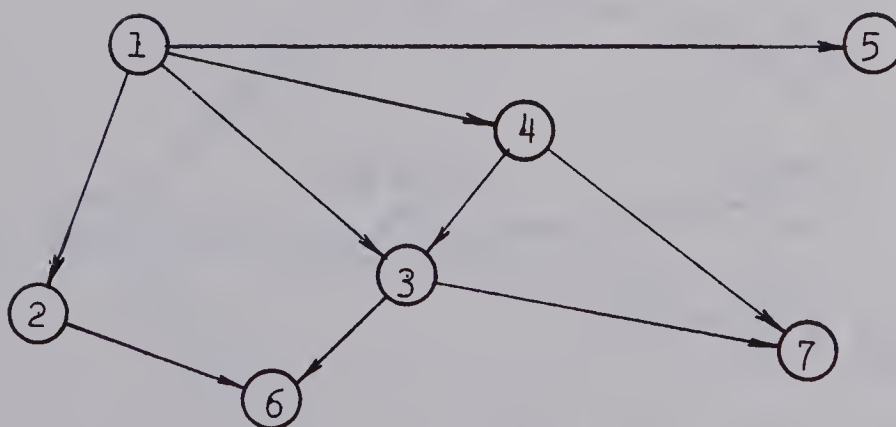
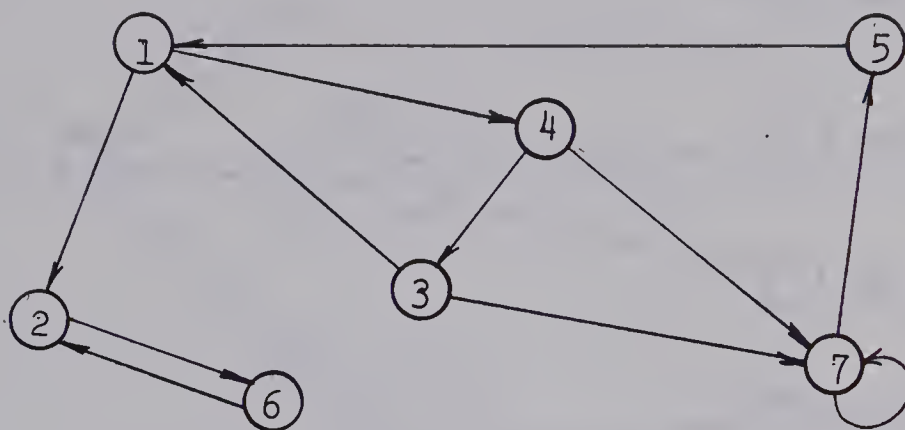


Figure 2.1 Directed Graphs, with and without Circuits

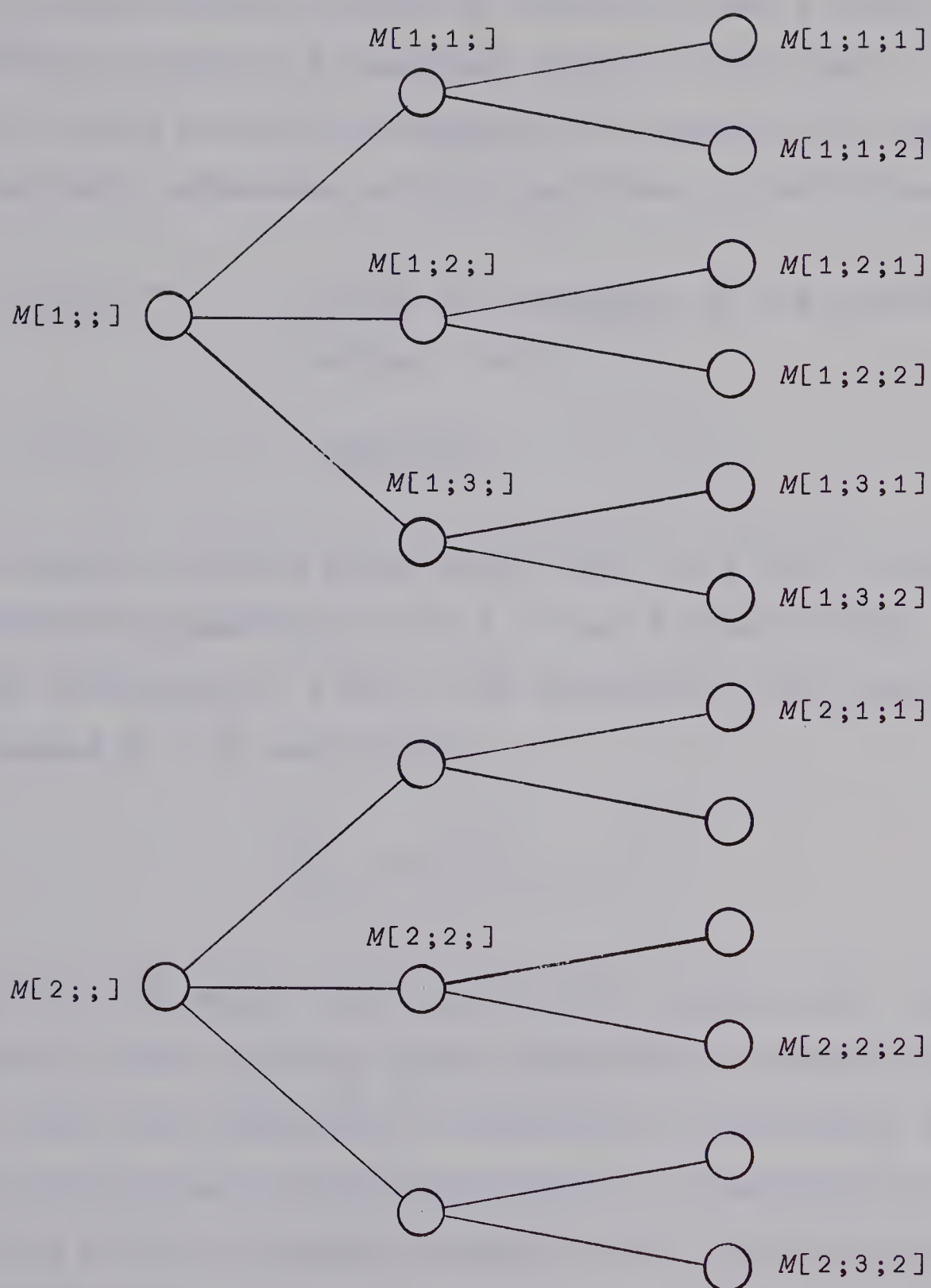


Figure 2.2 Doubly-Rooted Tree Representation
of Array M , where $t \wedge / (\rho M) = 2 \ 3 \ 2$

The association between an identifier and a field is recorded by means of a "masking" vector or "bit map". This vector, which we shall designate by U , contains the locations of the field referenced by the identifier in the following form:

$$\begin{aligned} \vdash U[I] &= 1 && \text{if the } I\text{th character of the record belongs} \\ &&& \text{to that field;} \\ \vdash U[I] &= 0 && \text{otherwise.} \end{aligned}$$

For example, assuming three identifiers A , B and C , with corresponding masking vectors X , Y and Z respectively, the fields referenced by B and C are subfields of the field referenced by A if and only if

$$\vdash \wedge / (Y \vee Z) = X \wedge Y \vee Z .$$

Figure 2.3 represents this relationship graphically. Each non-final node of either graph represents an identifier, each final node represents a character in the record, and a path exists from an identifier node to a character node if and only if that character belongs to the field referenced by that identifier.

Where duplicate identifiers are defined in different records, a qualifier must be used to distinguish between them. For example, ' B ' OF 2 is a qualified identifier referencing a

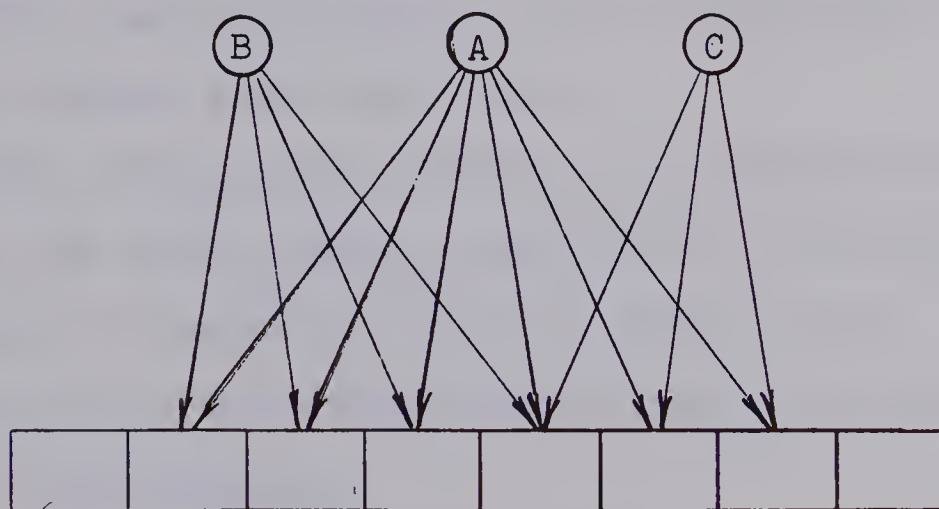
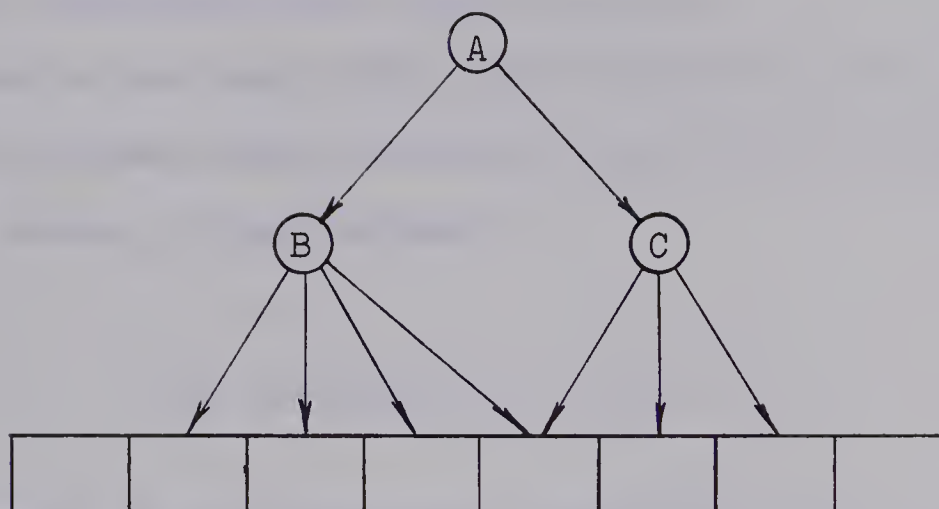


Figure 2.3 Two Equivalent Graphs of a
Data Structure

field in the second record area. An unqualified identifier is assumed to refer to the lowest-numbered record area for which that identifier has been declared.

Arrays of not more than three dimensions may be defined in DATA, provided that the entire array can be stored within a single record. The statement

```
25  ASSIGNΔTO 'A' SUB 2 1
```

is an example of an assignment statement which is valid if 'A' has been declared as a numeric array of suitable dimensions. *SUB* is a function of two arguments: the (unqualified) identifier and the vector of indices. The identifier refers to the entire array, and the indices select a particular component (i.e., one field) from that array.

In both of the above examples, the identifier refers to more than one field, and the addition of a qualifier or an index vector is needed to select a single field. Thus indexing (also called subscripting) may be considered to be a form of qualification.

A record can sometimes be accessed as a unit, without individual references being made to the fields within it. For example, the I/O statements employing the verbs *READΔINTO* and *WRITEΔFROM* can refer only to entire records, which they designate for transmission to or from *DATASET*.

A sequence of output operations will result in the arrival of a sequence of records at the storage device. This ordered group of records might be thought of as a vector of which each component is a record. More precisely, we shall assume that they form a tree in which each character of a record forms a leaf, and a root exists for each record. (See Figure 2.4.) In the special case where all records are of equal length, the tree becomes a matrix of characters, of which each record forms

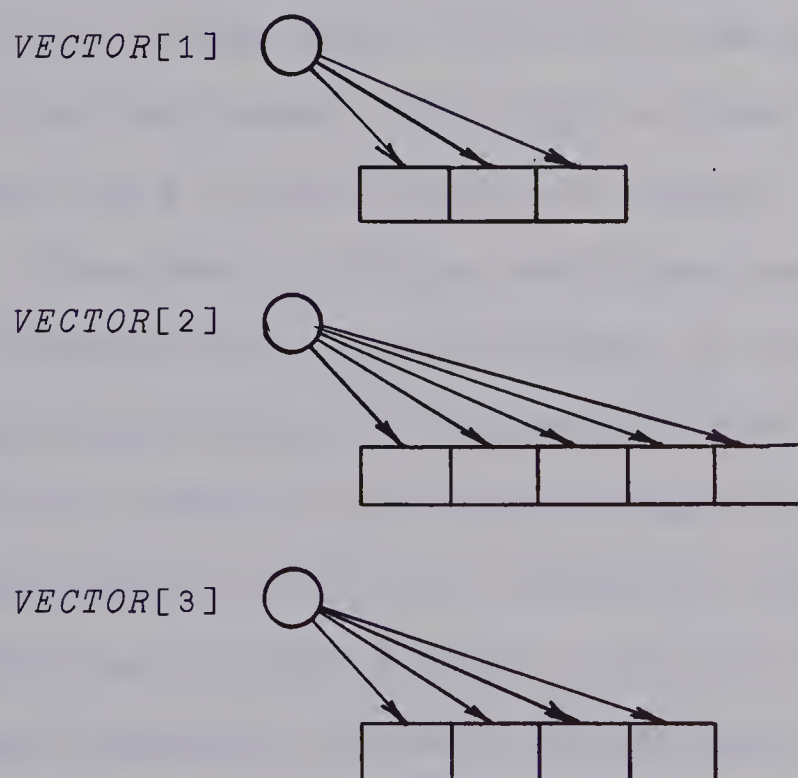


Figure 2.4 A Tree or Vector of Three Records

a row.

The storage device must represent this tree in a form which preserves the order and lengths of the individual records. It may be characterized as a second tree, of which each leaf is a byte, onto which the record tree will be mapped. The mapping of characters onto bytes will be one-to-one by definition. However, the mapping of records onto blocks may in principle, be many-to-one, one-to-one or one-to-many. That is, a block may contain the representation of more than one record, exactly one, or only a portion of a record, depending upon such factors as record lengths, physical characteristics of the storage device, and (to some extent) the preferences of the programmer. The DATA storage device *DATASET* is restricted to one record per block.

The tree of blocks and bytes must in turn be represented by *DATASET* which is, in effect, a vector of bytes. This can be accomplished by concatenating the blocks in order, forming a vector which can be encoded upon *DATASET*, provided that the locations of the block boundaries are recorded as well. One method of recording these locations is by specifying a particular character called a record mark, which will appear at the end of every block but in no other position. The record mark obviously cannot be a member of the alphabet of any data representation. The method which is used in DATA is suggested by Iverson's discussion of the representation of structured

operands in memory (Iverson (1962)). The locations of the representations of a tree may be found by reference to a "grid" matrix of two columns, where the I th row of the matrix contains the number of leaves connected to the I th node and the location of the first of these leaves. Where the leaves are ordered with no gaps between them, only a single column is required. If a vector X is constructed such that $X[I]$ contains the number of leaves connected to the I th node, the address of the first of those leaves will be one greater than the cumulative sum of the first $I-1$ components of X , that is $+1, X[1:I-1]$. A further simplification is possible with *DATASET*. Since *DATASET* may be accessed only sequentially, and since the rest position of the read/write "head" is always the initial byte of a block, it is only necessary to store in the initial byte of each block an integer specifying the number of bytes in the remainder of that block. Thus the information required for the retrieval of the next* block is always available, and every block is one byte longer than the record it contains. We shall call the initial byte of any block the control byte.

An input operation is simply the reverse of the output. The next block is retrieved from *DATASET* and transmitted to

* Section 7.1.3 of "Fortran vs. Basic Fortran" (Anon, 1964) defines the next record of a sequential file. The term "next" is used here in that same sense.

CORE. The record which it contains is transferred to the specified record area of *CORE*, and the position of the read/write head is advanced to the control byte of the subsequent block.

CHAPTER III

INPUT AND OUTPUT

3.1 Record-Oriented and Item-Oriented Input and Output

For any programming language statement which initiates an I/O operation, there exists a characteristic data unit, the smallest entity which that statement can designate for transmission to or from a data set. Blocks, records and fields exist in every case, with the relationships described in Section 2.1. Where a statement designates a record to be transferred to the output buffer, or from the input buffer, that statement will be described as record-oriented (a term coined by the designers of PL/1). Where a statement designates one or more fields (i.e., data items) for transfer, the statement will be described as item-oriented. (Use of the term "field-oriented" has been avoided because it is often used to refer to a mode of memory organization (e.g., Flores (1966)). Where all external representations are to be BCD, for example where the external data set originates as a deck of cards or where it will ultimately be printed, either of these two modes of I/O can be used. The choice of one of them will be made on one of the following bases:

1. Some programming languages are capable of only item-oriented or only record-oriented I/O. For example, if the program must be written in Fortran, then item-oriented I/O must be used.

2. Item-oriented I/O can usually be specified more concisely than record-oriented, but requires more execution time. Thus item-oriented I/O is used where it is desirable to save programming time at the expense of execution time, and record-oriented I/O is used where the converse is true.

A discussion of these two modes of I/O, in the DATA language and in other programming languages, is the subject of this chapter.

3.2 Item-Oriented I/O of DATA, Fortran and PL/1

With item-oriented I/O, the external representation is always BCD. Where the representation in memory is other than BCD, a data conversion is required. A data conversion is a transformation of the syntax of the representation which, ideally, does not change its semantic content. That is, the same number is represented (in a different form) before and after the conversion. In fact, however, a semantic transformation usually occurs as well, either through intentional rounding or truncation, or as a result of an error introduced by a radix conversion. (Matula (1968)).

An item-oriented I/O statement of the DATA language consists of one of the verbs *GET* and *PUT* followed by a list of identifiers. (Statements of the forms:

ITEMIN \leftarrow *I*
ITEMOUT \leftarrow *J* ,

(where $t \wedge (I, J) \in 13$) must be executed before the first execution of a *GET* or a *PUT* respectively, to specify which data sets are to be accessible by *GET* and *PUT* statements.) Figure 3.1 illustrates the effect of execution of a sequence of *PUT* statements. When the statement *PUT 'EB'* is executed, for example, the field in *CORE* referenced by 'E' is accessed and the contents are converted to BCD and extended with blanks on the right and left. If the output buffer is not already full, this BCD field is transferred into it. If the buffer is full, its contents are transmitted, as a record, to *DATASET*, and the field is transferred into the leading byte positions of the empty buffer. These steps are repeated for 'B' and for each subsequent identifier which appears as the operand of a *PUT* statement. The order in which data items appear in *DATASET* depends only upon the order in which their identifiers appear in *PUT* statements, not upon their relative positions in *CORE*. Execution of the statement *CLOSEFILES* after the final *PUT* statement will cause the remaining contents of the output buffer to be transmitted as the final record.

Figure 3.1, with "*PUT*" replaced by "*GET*", all arrows reversed and an arrow added from block number 3 to the buffer, illustrates the effect of a sequence of *GET* statements. Assuming the input buffer is initially empty, the execution of the statement *GET 'AFC'* will cause the next record from *DATASET* to be transferred into the buffer. The first group

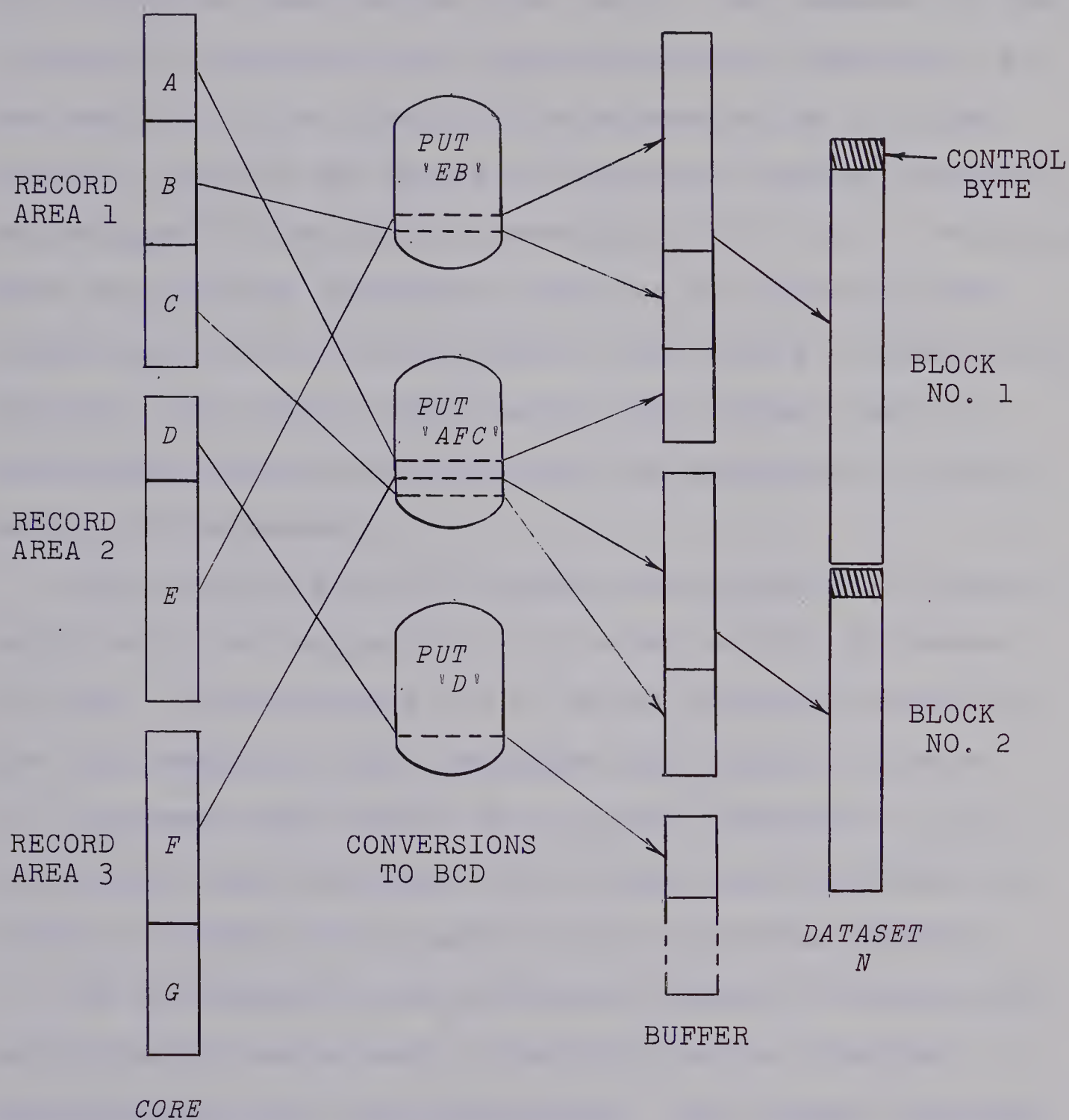


Figure 3.1 Item-Oriented Output in DATA

of non-blank characters followed by a blank or the end of the record is taken as the first field. The contents of that field are converted to the type declared for identifier 'A', and assigned to the field in *CORE* referenced by it. Subsequent fields in the buffer are similarly located, converted and assigned to the fields referenced by 'F' and 'C' respectively. When no non-blank characters remain in the buffer and the identifier list is not satisfied, a new record is transferred into it. Any fields remaining when the current list of identifiers has been satisfied will be accessible to a subsequent *GET* statement.

Both Fortran and PL/1 include item-oriented I/O statements which are very similar to the *GET* and *PUT* statements of DATA. (Item-oriented I/O is called "stream-oriented" in the PL/1 manuals.) Both languages also include a form of I/O statement which refers to a "format statement". (i.e., a statement which describes the external representation in detail. Formats are discussed in the following section.)

No alphanumeric type declaration exists in Fortran, but an "A" format may be used to specify that the internal representation is to be alphanumeric. The format indicates, on input, the number of characters from the data set which are to be stored in each word of memory, and, on output, the number of characters from each word which are to be transmitted. Thus the manipulation of alphanumeric data in Fortran

requires a detailed knowledge of the memory organization and data representations used by the particular computer. This is a serious disadvantage of Fortran, particularly since the necessary knowledge can rarely be gained from Fortran programming manuals, and must be learned by trial and error or consultation of more specialized manuals.

The greatest advantage of Fortran over PL/1 is that a format statement can be read as data. This is a useful feature in library programs, where the arrangement of the input data on cards may be different for each run. A PL/1 program can not read a format statement as data, but it can read a set of parameters from which the required format statement can be constructed.

3.3 Record-Oriented I/O - DATA, Cobol, PL/1

A record-oriented I/O statement of the DATA language consists of one of the verbs *WRITEΔFROM* and *READΔINTO*, with a record number and a data set number as its two operands. Figure 3.2 illustrates the effect of execution of a sequence of *WRITEΔFROM* statements. After execution of the statement *N WRITEΔFROM M*, for example, a block is constructed of which the control byte contains the unsigned binary fixed-point representation of the length (in characters) of the *M*th record in *CORE*, and the remainder contains that record. The block is then transmitted to *DATASET*.

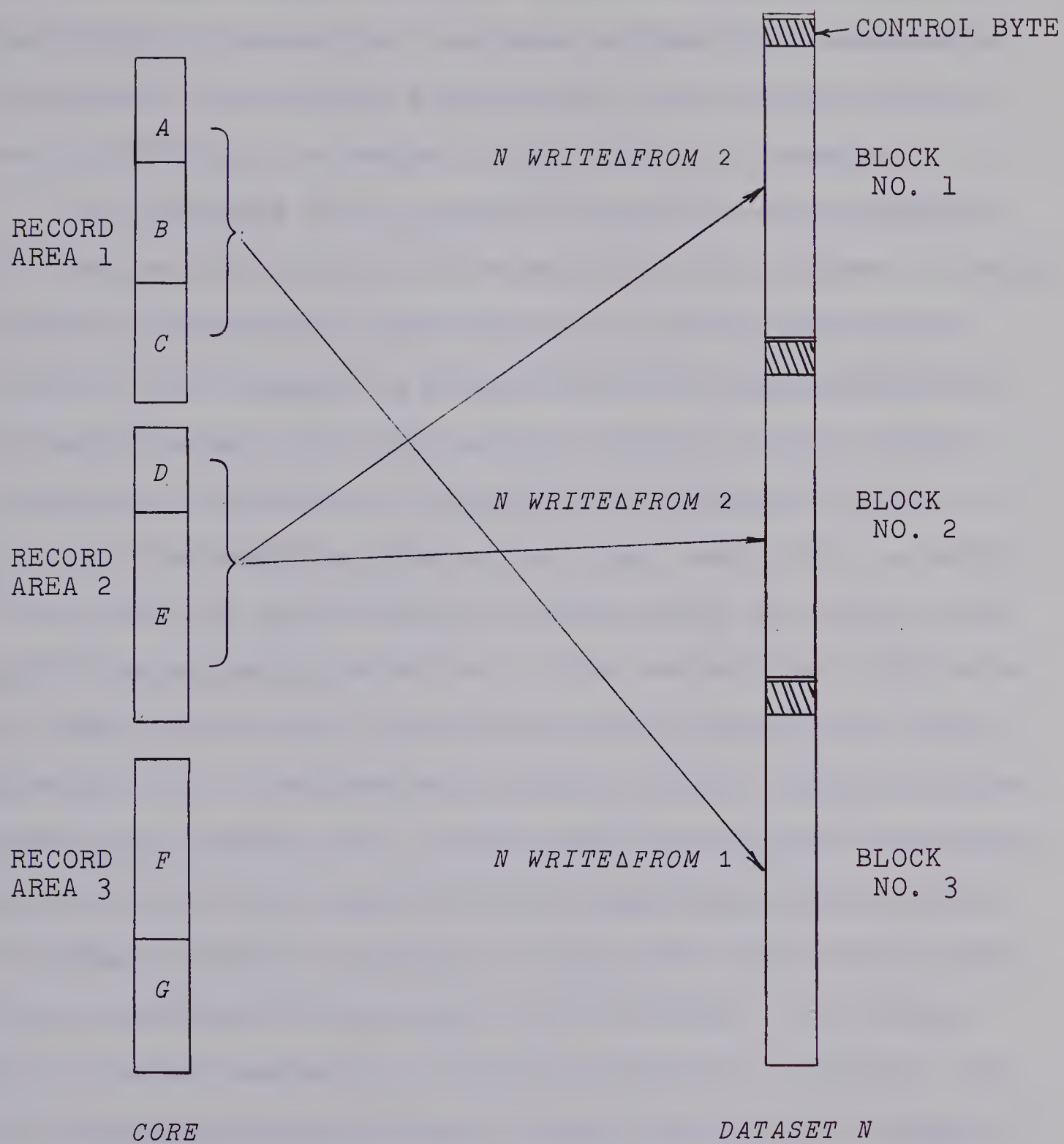


Figure 3.2 Record-Oriented Output in DATA

The same diagram with the arrows reversed illustrates the effect of executing a sequence of *READΔINTO* statements. The blocks are accessed sequentially, the control byte of each indicating the length of the record it contains.

As indicated in the previous section, record-oriented I/O can be used with a card reader or a line printer. However, a more representative application is in file maintenance. Consider, for example, a program which is used periodically to update a data set (also called a file, in this context) containing a collection of payroll or inventory records. It will use the existing data set as input and create as output a new data set containing the records which are input, plus additions and minus deletions. This new data set will serve as input for the next run of the update program, and also, possibly, for a program which prints monthly reports, statements, pay cheques, etc. Since only the computer can access this data set, the representations appearing upon it can be the same as those in memory, and many data conversions which would otherwise be necessary can be avoided. In a sense, this data set serves as a long-term portion of memory. If the computer and its external storage devices are thought of as a "system", the data transmissions initiated by this update program are not input and output, but simply manipulations of data within that system.

Another example of the transmission of data without con-

version between memory and an external data set is the swapping of APL workspaces. A limited number of workspaces can reside in memory at any time. The other "active" workspaces, if any, are stored on disk while waiting their turn to be processed. Corresponding to the control byte of the data record is a control block containing the information needed to identify the workspace, to retrieve it from disk, and to resume processing of it. In this case also, the external storage device serves as an extension to memory, and swapping takes place entirely within the "system". (Breed and Lathwell (1967)).

3.4 I/O Conversions

We have already stated that item-oriented I/O of numeric data includes a data conversion from an internal representation to BCD, or vice versa. The conversion on input is many-to-one, because many different arrangements of characters (digits, blanks, decimal point, etc.) can represent the same number. The essential information about a BCD representation is:

1. Field width and location - number of characters in the representation.
2. Default number of decimal places, to be assumed if no decimal point appears in the representation.
3. Scale factor, if any. Sometimes the internal and external representations differ by a factor which is a power of 10.

The output conversion is one-to-many. The information which is required for a complete specification of a choice from the "many" is:

1. Field width and spacing, vertically and horizontally.
2. Choice between fixed-point and floating-point representations.
3. Number of significant digits and/or decimal points.
4. Locations of signs and editing characters, if any.
5. Choice between truncation and rounding.
6. Radix of the representation - decimal, binary, octal, etc.

These choices are sometimes determined by the implementation. For example, on input using the Fortran NAMELIST or the PL/1 GET LIST or GET DATA options, fields are delimited by blanks or commas, and scale factors and default positions of the decimal point are not permitted. On output using these options, there is a one-to-one correspondence between external and internal representations. An internal short binary floating-point representation, for example, will be converted on output to a decimal floating-point representation with an arbitrarily fixed number of significant digits and width of field.

As another example where the programmer can not specify the external representation he will receive, an APL output representation is determined by the number represented, not

by its internal form. That is, there is a one-to-one correspondence between the number represented and the sequence of digits and characters printed. The integer 3, for example, is always printed as "3", never as "3.00" or ".3000000E1", even if its internal representation is floating-point.

In nearly all programming languages, however, the programmer is able to describe the desired external representations by means of a format statement. A format statement contains one or more format items, each of which describes the syntax of a BCD representation. The set of rules by which the description is encoded will be called a format language.*

(Where record-oriented I/O is used and all external representations are BCD, the type declaration statements perform the functions of a format language.)

For a given format item, i.e., a given syntactic description of the external representation, the data conversion required depends upon:

1. The number to be transmitted.

* Perlis was apparently the first to use this term. Perlis (1964).

2. The type declaration assigned to the internal representation, and the memory organization parameters which affect its meaning. For example, the Fortran type declaration INTEGER specifies a binary fixed-point representation occupying one word. Its length is 32 bits on the IBM 360/67, or 36 bits on the IBM 7040.

CHAPTER IV

SYNTAX-DIRECTED ANALYSIS OF FORMAT LANGUAGES

4.1 Introduction

The concept of a format "language", for which syntactic and semantic rules exist, has been used by Knuth et al. (1964) and by Perlis (1964) to describe their proposed format conventions. The use of rules of syntax in processing a language statement is called syntax-directed analysis. The process of analyzing a format item (or a string of symbols in any language) to discover which rules of syntax were used to form it is called parsing. Any notation which may be used to specify the rules of syntax of a format language is called a metalanguage. The most commonly-used metalanguages are derived from Backus Normal Form (BNF, which first appeared in the Algol 60 Report). We shall use the variant suggested by Iverson (1964).

This chapter describes a system of APL functions which use a transition diagram representation of the rules of syntax of a format language as a basis for translating format items of that language into a universal format language (abbreviated UFL). The entire chapter refers to that system, unless otherwise indicated.

4.2 A Parsing Algorithm

This section describes both the preparation of a format language for syntax-directed analysis, and the parsing process. However, most of the description could be applied to the analysis of any language.

Any character of the string being parsed is called a terminal symbol. A syntactic unit is any group of symbols defined by a rule of syntax. It may contain terminal symbols or other syntactic units, or both. A transition diagram is a directed graph having one initial node and one or more final nodes. Each arc of the graph corresponds to one of:

1. a syntactic unit
2. a terminal symbol
3. a blank path.

Each transition diagram represents one or more rules of syntax. A simple example of a format language, a subset of that of Fortran, is shown in Figures 4.1 to 4.3. The BNF representation of the rules of syntax (Figure 4.1), the transition diagram representation (Figure 4.2) and the matrix *DSFTAB* (Figure 4.3) are equivalent. Columns of *DSFTAB* represent, respectively:

1. Diagram number.
2. Initial node, and
3. Final node of the arc represented by this row.
4. 0 denotes a terminal symbol, 1 denotes a syntactic unit.


```

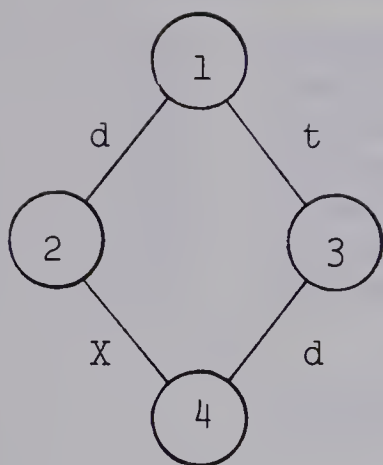
d  DIGIT:    1|2|3|4|5|6|7|8|9
t  TYPE  :    A|I
1  ITEM  :    d X|t d
2  LIST  :    1 | 1,*
3  FRMT  :    ( 2  )

TERMINAL SYMBOL:  123456789 AI X , ( )
TYPE:              4      5  6 7 8 9

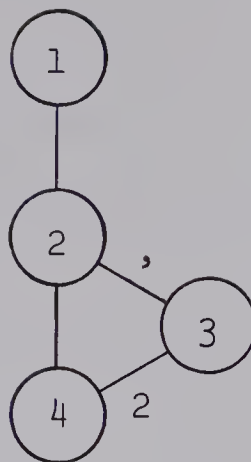
```

Figure 4.1 Syntax of a Subset of Fortran

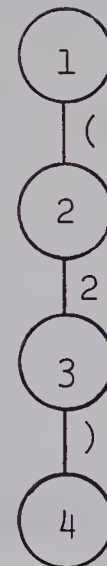
5. "Type" number of a terminal symbol, or Diagram number of a syntactic unit, or 0 denoting a blank path.
6. 0 if this node is not an exit node, or a non-zero integer indicating which exit node it is.
7. An integer indicating to the semantic-analysis algorithm the action to be taken each time this node is reached, except that, for an exit node, this information appears in column 6 and column 7 contains 0.



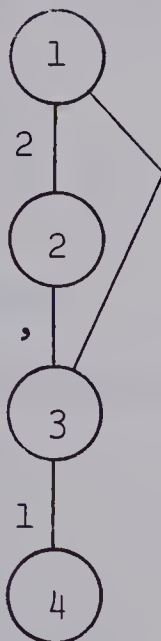
1 ITEM



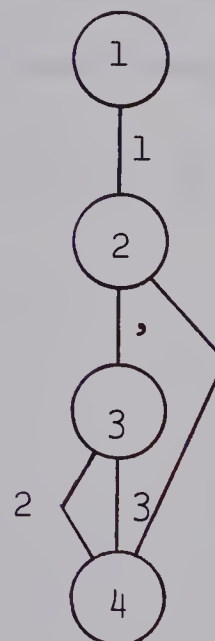
2 LIST



3 FRMT



2a (see text)



2b (see text)

Figure 4.2 Transition Diagrams, with
Node Numbers


```

      ▽ DSF STR
[1]    FORM←STR
[2]    TABLE←DSFTAB
[3]    CHARS←DSFCHARS
[4]    CHINDEX←DSFCH
[5]    TYPES←DSFTYP
[6]    LANGCODE←0
[7]    NEXTITEM
      ▽

      DSFTAB

      1  1  2  0  4  0  0
      1  2  4  0  6  1  0
      1  1  3  0  5  0  0
      1  3  4  0  4  1  0
      2  1  2  1  1  0  0
      2  2  3  0  7  0  0
      2  3  4  1  2  2  0
      2  2  4  0  0  2  0
      3  1  2  0  8  0  0
      3  2  3  1  2  0  0
      3  3  4  0  9  3  0

      DSFTYP
      ITEM
      LIST
      FRMAT

      DSFCHARS
      123456789AIX,( )

      DSFCH
      4  4  4  4  4  4  4  4  4  5  5  6  7  8  9  0

      PRINT←1

      DSF '(3X,I5,A3)'

      ITEM    3X
      ITEM    I5
      ITEM    A3
      LIST    A3
      LIST    I5,A3
      LIST    3X,I5,A3
      FRMAT    (3X,I5,A3)

```

Figure 4.3 Arrays for Subset of Fortran

Transition diagram 1 specifies that the syntactic unit *ITEM* may consist of a digit followed by an "X", or alternatively, an "A" or an "I" followed by a digit. The sample parse of Figure 4.3 contains examples of these three forms of *ITEM*.

Diagram 2 of Figure 4.2 defines a *LIST* as a concatenation of one or more *ITEMS* separated by commas. Although this definition is recursive, it will terminate after a finite number of entries to the diagram because terminal symbols from the string are identified upon each entry. The blank path from nodes 2 to 4 implies that if a match is not found on any other path from 2, an exit is allowed. A blank path is always tested last, after all other paths having the same initial node have failed to produce a match. Therefore no more than one blank path may begin on any node.

Diagram 3 completes the specification of the language subset, and is the starting point for a parse. Diagram 2a is equivalent to diagram 2 when the parse is performed manually. However, our rule of testing a blank path after all nonblank paths from the same node results in an infinite number of entries to the diagram, since its first arc initiates a new entry without first matching a terminal symbol. A human parser would (probably) notice this and choose the empty path at the appropriate times but an algorithm cannot. In diagram 2b, the addition of a single arc to diagram 2 (and therefore,

a single row to the matrix) extends the language subset to include constructions such as: (A3,(3X,A4)) and (2X,(A3,I5)), which contain one *FORMAT* within another.

Parsing proceeds under control of the rules of syntax represented by *DSFTAB* (Figure 4.3), beginning at the row corresponding to the first arc of diagram 3. Each time a syntactic unit is encountered, the current row index is pushed down on a stack and a new diagram (a new row) is entered. When a diagram is either traversed satisfactorily or abandoned because no match could be found, the stack is popped up to return control of the previous diagram. When the stack becomes empty, the parse is complete or else an error has been found. If one or more matches have already been found within a diagram but no match can be found for the current symbol of the format string, (i.e., no exit node can be reached) an error has been encountered. Because of this requirement, (the "no back-up" rule, Conway (1963)), the designer of a transition diagram must ensure that the first terminal symbol of any syntactic unit will fail to be matched within a diagram, unless the entire syntactic unit can be matched. For the simple language illustrated in Figures 4.1 to 4.3, this is easily accomplished. In general, however, the no-backup rule must be carefully observed. For example, since the initial terminal symbol of each of the Fortran format items 2X, 4A2, 3HXYZ, and 5(2X,I3) is an integer, all must be represented by the same transition diagram. That diagram may, however,

have as many distinct exit nodes as necessary. (See Figure C.1, Appendix C.)

The global variable *NUMSTACK* ("number stack") stores integers which have been identified by the parsing algorithm but have not yet been used in interpretation of the format item. The function *TOPNUM* removes from *NUMSTACK* the integer which was stacked last, returning it as a result.

The parsing algorithm *NEXTITEM* calls the function *SEMANTICS* (Figure 4.4, statement 14 or 21), which controls the interpretation of a format item. When any arc of a transition diagram is matched, whether by matching of a terminal symbol or by successful traversing of another transition diagram, a non-zero value in column 6 or 7 of the row of *TABLE* which corresponds to that arc indicates which interpretation rule is to be executed. (Zero indicates that no action is required.) For an exit node, the argument to be passed to *SEMANTICS* is found in column 6 of *TABLE* (*NEXTITEM*, statement 21). Otherwise (statement 14), column 6 contains 0 to prevent an exit, and column 7 supplies the argument.

For example, during a parse of the PL/1 format item 2E(14,8), *SEMANTICS* is called six times with nonzero arguments. Each integer is stacked as it is identified. Matching of the "E" causes initialization of a counter, which is incremented and tested at each subsequent matching of a ",". If the counter exceeds its limit, 2 for an E format item, an error

▽ NEXTITEM;ROW;I;R;TEST;DIAG;MK;A

```

[1]  R←1ROWPT←0
[2]  ROW←TABLE[;1]1/TABLE[;1]
[3]  IDS←,SYPTR←1
[4]  TERMM:→SKIP 4×~TABLE[ROW;4]
[5]  ROWPT←ROWPT,ROW
[6]  ROW←TABLE[;1]1TABLE[ROW;5]
[7]  IDS←IDS,SYPTR
[8]  →TERMM
[9]  →SKIP 5×0=TABLE[ROW;5]
[10] →(TABLE[ROW;5]≠CHINDEX[CHARS1FORM[SYPTR]])/BKUP
[11] →SKIP1TRACE
[12] ' MATCHED: ',FORM[SYPTR]
[13] SYPTR←SYPTR+1
[14] SEMANTICS TABLE[ROW;7]
[15] →(0=TABLE[ROW;6])/GOO
[16] I←IDS[ρIDS]-1
[17] DIAG←TABLE[ROW;1]
[18] →SKIP~PRINT^TABLE[ROW;6]∈1(ρTYPES)[1]
[19] (,TYPES[TABLE[ROW;6];]),' ',FORM[I+1-1+SYPTR-I]
[20] →SKIP 0≥TABLE[ROW;6]
[21] SEMANTICS TABLE[ROW;6]
[22] IDS←IDS[1-1+ρIDS]
[23] →(0=ROW←TOP)/0
[24] →SKIP 8×DIAG=TABLE[ROW;5]
[25] →SKIP 1
[26] GOO:→(0=MK←GO ROW)/BKUP
[27] ROW←MK
[28] →TERMM
[29] BKUP:TEST←(^[2]((ρTEST)ρTABLE[ROW; 1 2])=TEST←TABLE[
; 1 2])/1(ρTABLE)[1]
[30] TEST←,(TEST>ROW)/TEST
[31] →SKIP 2×0=ρ,TEST
[32] ROW←1ρTEST
[33] →TERMM
[34] ROW←TOP
[35] →SKIP 2×SYPTR>IDS[ρIDS]
[36] IDS←IDS[1-1+ρIDS]
[37] →(0<ρ,ROWPT)/BKUP
[38] ' WRONG SYNTAX. ',FORM
[39] ((17+SYPTR)ρ' '),'+!

```

▽

▽ R←GO RO;TE;I;J

```

[1]  TE←(TE>RO)/TE←(TABLE[;1]=TABLE[RO;1])/1(ρTABLE)[1]
[2]  →(~TABLE[RO;3]∈TABLE[TE;2])/R←0
[3]  R←TE[TABLE[TE;2]1TABLE[RO;3]]

```

▽

Figure 4.4 Parsing Algorithm

message is printed. Matching of the ")" causes interpretation of the format item to begin. The counter indicates how many integers are to be unstacked for the interpretation. The replicator "2" must be left as the "top" element of *NUMSTACK* when interpretation of E(14,8) is completed.

Since a set of transition diagrams incorporates the rules of syntax of a format language, any string of symbols which cannot be parsed on the basis of those diagrams contains a syntax error, by definition. A string of symbols which satisfies the rules of syntax but does not constitute a valid format item is said to contain one or more semantic errors. The set of rules of syntax for any format language, however, is not unique. A comparatively simple set can be chosen which will correctly parse any valid format item, but will allow many invalid ones to pass undetected. A more complex set of rules can be chosen which will detect all except a few classes of invalid format items. The distinction between syntactic and semantic errors is thus, to some extent, an arbitrary one. An example of an error which is clearly semantic is one in which the values of digits are invalid, as in the Fortran format item F2.9. An example of an error which may be treated as either syntactic or semantic is the PL/1 format item P'Z9/Z9/Z9'. Since the "Z" indicates suppression of a leading zero, it may never follow a "9", which indicates a printed digit. The UFL system treats this as a semantic error,

although as a question of "the permissible arrangements of symbols", it should perhaps be classified as syntactic. The syntax rules represented by *PLTABLE* accept any sequence of "Z's" and "9's", and the semantic-analysis function *COMPL1* (statements 34, 35 and 36) prints an error message if a "Z" appears anywhere following the first appearance of a "9". Transition diagrams could probably be drawn which would detect this error, but the additional complexity does not appear to be justified. In the current versions of *COMPL1* ("compile PL/1") and *COMPAFORT* ("compile Fortran"), a syntactic error terminates the parse but a semantic error does not. (Figures 4.5 and 4.6.)

4.3 An Experimental Universal Format Language

The semantic-analysis algorithm, under control of the parsing algorithm, translates each format item into an equivalent "target language" representation which is independent of the format language originally used. This target language must be capable of specifying fully any data conversion which might be required. However, it need not be concise or easily coded by hand, since the translation will always be done automatically. The target language consists of two integer matrices, *FORMM* and *TARGMAT*. *FORMM* serves as the index to the representation, since each column (with minor exceptions) corresponds to one format item and points to the appropriate portion of *TARGMAT*.


```

          PRINT←1
          FORTFORM '(F5.1,3(A4,2X,I3))'
SAVE CODE?
NO
DIGIT 5
DIGIT 1
CODE F5.1
ITEM F5.1
DIGIT 3
DIGIT 4
WIDTH A4
ITEM A4
DIGIT 2
LIT 2X
DIGIT 3
WIDTH I3
ITEM I3
LIST I3
LIST 2X,I3
LIST A4,2X,I3
FORM (A4,2X,I3)
REPFM 3(A4,2X,I3)
LIST 3(A4,2X,I3)
LIST F5.1,3(A4,2X,I3)
FORM (F5.1,3(A4,2X,I3))

          FORTFORM '(F5,1,3(A4,2X,I3,))'
SAVE CODE?
NO
DIGIT 5
WIDTH F5
ITEM F5
DIGIT 1
      WRONG SYNTAX. (F5,1,3(A4,2X,I3,))
                        ↑

```

Figure 4.5 A Parse of a Fortran Format
Statement


```

PRINT←1

      PLFORM '(3F(9,5,6),P'ZZ9,Z99')'
SAVE CODE?
NO
INT    3
INT    9
INT    5
ILLEGAL COMMA: (3F(9,5,6),P'ZZ9,Z99')
                        ↑
INT    6
SPEC    6
SPEC    5,6
SPEC    9,5,6
ITEM    F(9,5,6)
ZEDS    Z
ZEDS    ZZ
SEP
INT    9
SEP
SEP    ,
ZEDS    Z
SEP
INT    99
SEP
GRP    99
GRP    Z99
GRP    9,Z99
GRP    ZZ9,Z99
ILLEGAL SEQUENCE: P'ZZ9,Z99'
                        ↑
LIST    P'ZZ9,Z99'
LIST    3F(9,5,6),P'ZZ9,Z99'
FORM    (3F(9,5,6),P'ZZ9,Z99')

```

Figure 4.6 A Parse of a PL/1 Format
Statement

For example, assume that the data item A is to be converted for output by reference to a target language description, and that the corresponding column of *FORMM* has the values:

<u>Value</u>	<u>Meaning</u>
10	Decimal, i.e., a numeric value
0	Truncate. (1 would indicate rounding).
0	Right justified in the field (1 would indicate left justification).
2	Replicator - this format item is to be used for two consecutive data items.
5	Field width of BCD representation.
K	A location in <i>TARGMAT</i> .

We shall call this column the vector *M*. A matrix *Y* will also be required, specified by

$$Y \leftarrow TARGMAT[;M[6] + 1:M[5]]$$

Let us assume that *Y* has the value

0	0	0	0	0
1	1	0	0	0
0	0	0	1	0
2	2	0	0	0
0	0	0	0	0

These values of M and Y correspond to the PL/1 format item 2F(5,1). Their interpretation is as follows: Each column of Y describes a single character of the desired BCD representation. The 1 in $Y[;3]$ marks the position of the decimal point. The 2's in $Y[;4]$ mark possible sign positions, blank for plus and "-" for minus. Thus the number of character positions available for digits and sign is $+/Y[3;]=0$. (In general, the number is $+/Y[3;]\leq 0$, since $Y[3;]$ may contain a negative value indicating the position of an assumed decimal point which is not printed.) Where A is positive there may be 4 digits, since no sign will be printed.

A is now converted to a character string of digits, decimal point, sign, etc. The action to be taken on overflow, that is, where too few character positions have been provided to the left of the decimal point, is indicated by the contents of $Y[1;]$. In this example, $Y[1;]$ contains only zeros, implying that the sign and leftmost digits may be truncated if necessary, the remaining digits being printed. Any nonzero elements of $Y[1;]$ are used as subscripts in the vector *CHARSET* to determine which characters should replace the corresponding digits of the representation. For example, where

$$\dagger \wedge /Y[1;] = CHARSET \text{ ' '*}$$

the field will be filled with asterisks if a sign or leading

nonzero digit has been lost (as in Fortran).

Some values of *A* and the representations to which they would be converted by the matrix of this example are:

<u>Value</u>	<u>Representation</u>
+3.38	3.3
+33.87	33.8
-33.87	-33.8
-338.70	338.7
-3387.00	387.0

Sample parses of format statements of Fortran and PL/1 are shown in Figures 4.5 and 4.6. The syntactic units identified within a format statement are listed or not depending upon whether the value of the global variable *PRINT* is 1 (Figures 4.5 and 4.6) or 0 (Figure 4.8), respectively. By typing "NO" in response to the message "SAVE CODE?", the programmer indicates that previous entries (if any) in *FORMM* and *TARGMAT* are to be overwritten. "YES" causes subsequent entries to follow the previous ones, i.e., to occupy succeeding columns.

As a second example, consider the Fortran format (3X,4(3F13.4,I7)). The nested parentheses demonstrate the flexibility of the transition diagram method of analysis.

Translation proceeds as follows:

After the item 3X has been identified and translated into a column of *FORMM*, the integer 4 is identified and pushed down on *NUMSTACK*. Next, the "(" is matched, and the current value of the variable *FNEXT* (i.e., the index of the next available column of *FORMM*) is pushed down on *PARENSTACK* as a record of the location of the first format item within parentheses. The integers 3, 13 and 4 are identified in turn and pushed down on *NUMSTACK*. The format item F13.4 can now be coded in *TARGMAT*. The node by which an exit is made from transition diagram number 3 returns a value of 3, transferring control to the statement of *COMPΔFORT* labelled "THREE". The 4 and 13 are unstacked (in that order), and they and the "F" are used in construction of *FORMM*[;*FNEXT*] and the corresponding 13 columns of *TARGMAT*. When the scan of the format item F13.4 is completed, the scan of the "RPITM" (i.e., "replicated item") 3F13.4 is thereby completed also, and the next integer from *NUMSTACK*, 3, is assigned to *FORMM*[4;*FNEXT*], overwriting the 1 which was placed there as a default replicator. *FNEXT* is incremented, I7 is translated and *FNEXT* is incremented again. The next terminal symbol, the ")", indicates that a *FORM* has been traversed, thereby completing the traverse of a *REPFM* ("replicated format"), which returns a value of 13.

This value is passed to the function *COMPΔFORT* to indicate that the top entry of *NUMSTACK* is a replicator for

the entire *FORM*. A dummy vector (i.e., a column which does not refer to any one format item) is entered in *FORMM*:

```
COMPΔFORT[29] THIRT : FORMM[;FNEXT] ← 0 0 0, TOPNUM, 0, TOPPAREN
```

Function *TOPNUM* unstacks the top entry from *NUMSTACK* (i.e., the replicator 4), and function *TOPPAREN* unstacks the top entry from *PARENSTACK*, which is the column index in *FORMM* which corresponds to the first format item within this pair of parentheses. Where there is no replicator for a *FORM*, the value 14 is passed to *COMPΔFORT* to indicate that the top entry of *PARENSTACK* should be unstacked and discarded:

```
COMPΔFORT[31] FORT: → ρ0/TOPPAREN
```

Parentheses nested to any depth can be interpreted by the UFL system, as long as the number of replicated formats (*REPFM'S*) does not exceed the number of rows in the replicator table *REPTAB* (a local variable of *WRITEOUT*, Figure 4.7). In practice, however, it is difficult to find an example of a format statement which requires more than two levels of nesting.

WRITEOUT makes use of *TARGMAT* and *FORMM* and controls the conversion of data items. Upon exit from the function, it returns a character vector consisting of the representations of all the components of "*LIST*". The logic of *WRITEOUT* is


```

▽ STR←WRITEOUT LIST;I;REPTAB;REP;ST;J;K;X;Y;Z

[1]  STR←⋈I←0
[2]  LOOP:→SKIP 1<I+1+(FNEXT-1)|I
[3]  REPTAB← 7 2 ρ0
[4]  →SKIP 3×FORMM[5;I]≠REP←0
[5]  →(FORMM[4;I]≤K←REPTAB[J←(REPTAB[;1]∈0,I)⋈1;2]+1)/LOOP
[6]  REPTAB[J;]←I,K
[7]  I←FORMM[6;I]
[8]  →SKIP 2×FORMM[1;I]≥1
[9]  STR←STR,FORMM[5;I]ρ' '
[10] →LOOP
[11] →SKIP 2×0≠FORMM[4;I]
[12] STR←STR,CHARSET[TARGMAT[1;FORMM[6;I]+⋈FORMM[
    5;I]]]
[13] →LOOP
[14] STR←STR,FORMM[;I]OUTITEM 1ρLIST
[15] →SKIP 2×0=ρLIST←1 SUFFIX LIST
[16] →SKIP -1×FORMM[4;I]>REP←REP+1
[17] →LOOP
▽

```

Figure 4.7 "WRITEOUT"

summarized below.

<u>Statement</u>	<u>Effect</u>
2-3	The first <i>FNEXT</i> -1 columns of <i>FORMM</i> correspond to the format items to be used. <i>I</i> increases cyclically, never exceeding <i>FNEXT</i> -1.
4-5	<i>FORMM</i> [5; <i>I</i>] is normally a field width. It can be zero only if the <i>I</i> th column represents a replicator. The value of the replicator (row 4) is compared to <i>REPTAB</i> [<i>J</i> ;2], which indicates how many times the parenthesized portion has been traversed.
6-7	If the parenthesized list is to be executed again, the incremented value <i>K</i> is stored in <i>REPTAB</i> [<i>J</i> ;2] and <i>I</i> is reset to the column corresponding to the first format item of that list.
8-9	If \neq <i>FORMM</i> [1; <i>I</i>]=0, spaces are to be entered (e.g., FORTRAN "3X").

<u>Statement</u>	<u>Effect</u>
11-12	If $\pm FORMM[4;I]=0$, a literal (Hollerith) field is indicated. <i>TARGMAT</i> [1;] contains the indices which indicate the contents of the field.
14-15	A representation is catenated onto <i>STR</i> . Exit if the list is empty.
16-17	Skip backwards to statement 14 if the replicator for this format item is not satisfied. Otherwise, branch to statement 2, to increment <i>I</i> .

4.4 Evaluation and Possible Future Extensions

The system of algorithms called UFL is designed to analyze, and to translate into a "universal" target language, any statement from a format language for which the rules of syntax and semantics have been specified in a suitable form. Reasonably complete subsets of the format languages of both Fortran and PL/1 have been analyzed in this manner, demonstrating the feasibility of:

1. Syntax-directed analysis of format languages.
2. A generalized format language, into which other format languages can be translated.

The principal features of these two format languages which can not yet be processed by UFL are:

1. Carriage-control format items.
2. Any input format.
3. PL/1 format items where one or more of the integers specifying the replicator, field width, number of decimal places, etc. are replaced by arithmetic expressions to be evaluated at execution time.

The first two of these limitations result from the lack of an external data set upon which the representations described by a format item could be stored. The present version of *WRITEOUT* (Figure 4.7) constructs and displays a character string containing the representations specified by the portion of *TARGMAT* which is being used as an output format. This character string could be encoded upon *DATASET* if the DATA and UFL systems could be combined, and input operations between *DATASET* and *CORE* could also be performed under format control. Carriage control format items could be defined as well. The third limitation could also be partially overcome by combining DATA with UFL. A DATA identifier (but not an arithmetic expression) could be allowed to appear in a PL/1 format item to be evaluated at execution time. Clearly, the UFL notation would have to be extended to allow this flexibility, but the basic organization would not have to be changed.

Ideally, a generalized format language should be general


```

PRINT←0

PLFORM '(E(13,4,6),X(3),P' '----,999V.99' ' )'

SAVE CODE?
NO

WRITEOUT ~25.9 ~25.9

-25.9000E+00      - .025.90

WRITEOUT 1.222E8 1.222E8

12.2200E+07      2200,000.00

WRITEOUT 18.278 18.278 18.278

18.2780E+00      018.27  18.2780E+00

FORTFORM '(2X,' 'START' ' ',2X,I3,E13.4)'

SAVE CODE?
NO

WRITEOUT ~13 234E~9

START  -13    0.2340E-06

WRITEOUT ~137 ~137 24

START  ***   -0.1370E 03  START    24

```

Figure 4.8 Sample Output

enough to represent any format item which will ever submitted for parsing. That is, it should be possible to freeze the rules of the language without ruling out the possibility of translating any future format item into it. Although this ideal is not likely to be reached, it does suggest a direction for future work.

CHAPTER V

CONCLUSION

5.1 Languages

The usage of the terms language, syntax and semantics in this thesis is more general than usual. Data items, data structures and format statements are all represented in a computer memory in very similar forms. The system of assigning meaning to each of these three classes of representation has been referred to as a language, and each representation has been said to possess syntactic (structural) properties and semantic properties (associated with its meaning).

The necessity for a distinction in terminology between the structural and the computational (more generally the transformational) properties of data structures has been noted by Standish (1967), who suggests adoption of the terms "data structure" and "data type". For example, using this proposed terminology, a "class" of arrays would be called a data structure, while a class of arrays which can undergo a particular set of transformations would be called a data type. (That is, every data type is a subset of a data structure.) For a particular class of $N \times N$ arrays, some possible sets of transformations are:

1. Transposition, indexing, and tests for equality only.
e.g., an alphanumeric matrix.

2. The above operations, plus all arithmetic and logical operations. e.g., a numeric matrix.
3. The above operations, plus computation of the inverse, determinant and eigenvalues. i.e., a numeric matrix which represents a linear transformation, or the coefficients of a system of equations.

Exactly what is meant by a "class" is not made clear, but apparently the set of all 3×3 matrices, for example, would be called a data structure, and the subset of these for which eigenvalues exist would constitute a data type. These terms are well-chosen in that "structure" suggests the arrangement of the symbols of a representation, and "type" suggests inherent properties which are not necessarily structural. However, by defining each data type in terms of some data structure, we rule out the possibility of any structure-less entity such as a number. Any number can undergo addition and multiplication, whether it is represented in binary, decimal or some other radix (or is not represented at all). That is, the capacity to undergo arithmetic operations is independent of structure. It is a property of the number itself. However, a number is not a data type. Only particular representations of it, that is, particular data structures, can be data types.

A more satisfactory way of expressing the relationship

between a number and its representations* is to say that they have most semantic properties in common, for example a number or any numeric representation can undergo arithmetic operations. However, while a representation has structure and therefore has syntactic properties, a number has neither.

The linguistic terms were chosen because of this capacity to describe properties of an abstract quantity as well as those of its representations. The terms data type and data structure, as defined by Standish, tend to obscure this distinction. Furthermore, since the existing linguistic terminology is adequate for the description of representations, the introduction of new terms is not justified. The use of the same terminology in discussions of programming languages and data representation languages is highly desirable because of the basic similarities that exist between data and instructions stored in memory. For example, the semantic properties of a Fortran format statement which is to be processed as alphanumeric data include the properties of any alphanumeric data, i.e., the capacity to undergo assignment, tests for equality, etc., as well as the capacity to initiate a specific data conversion. This is a simple, almost trivial, example of a program modifying itself. (A capability which Perlis (1966) has suggested for incorporation into Algol.)

* For a mathematical approach to the representation of data, see Mealy (1967).

5.2 N-Dimensional Languages

We have frequently referred to a language as a system of assigning meaning to a "string" of symbols, and, on the other hand, we have referred to the syntax of a data structure. The objection might be raised that an array, for example, is a data structure which consists not of a string of symbols but of a rectangular block of symbols, and that the terminology of linguistics is therefore not appropriate. One answer to this objection is that, since any computer memory consists of a "string" of bytes, any structure which is to be encoded in memory must be represented as a string of characters. A more basic answer is that written languages need not be restricted to the form of a string. Two-dimensional arrangements of symbols are treated as languages in the study of graphic displays (see the "Non-Compiler Applications" section of McIntosh (1968)). Multi-dimensional languages might also prove useful at some future time.

5.3 Input and Output

Two forms of I/O, record-oriented and item-oriented, have been described here. They were first given names by the designers of PL/1, although they had existed for several years in Cobol and Fortran, respectively. Record-oriented I/O is usually used to transfer data between the high-speed memory of the computer and the slower, longer-term "memory"

of external storage. Item-oriented I/O is more interesting from the point of view of translation between "computer-readable" and "human-readable" representations. A remark by Flores (1966) suggests that any data which the computer can read should be called storage, and that data which it cannot read (e.g., hard copy) be called output. The processes which occur in both forms of I/O are essentially alike, the distinction being in the units of data to which the programmer has access. Fortran binary I/O is a borderline case which we have called item-oriented. However, the statements `WRITE (3)A` and `READ (3)A`, where `A` is an array, are in fact identical to record-oriented statements in every way, since a record is referred to by a single identifier and transmitted without conversion. (The array `A` occupies exactly one record only if its total length in bytes is equal to the record length parameter specified for the external data set.)

5.4 A Proposal for the Standardization of Format Languages

A format item describes the syntax of a single BCD representation. It is the smallest meaningful unit which a string of symbols can represent in a format language. A format statement, consisting of a list of format items, is used by a programmer to specify the arrangement and form of data representations required (on output), or expected (on input) on the external data set. The design of the format language

therefore affects the convenience with which a program can be written and the input data prepared.

The exchange of programs between computer installations is hampered by, among other things, the variations which exist between different implementations of any format language. For example, it is not uncommon for a Fortran compiler to employ one or more format items which other Fortran compilers do not recognize. Standardization would be desirable to ensure that all implementations of each format language were consistent in:

1. The set of permissible format items.
2. The representation described by any format item.
3. The action to be taken in exceptional cases, such as a conflict between the format item and the type declaration.

Ultimately, a single format language might be adopted for use by all programming languages which use item-oriented I/O, including Algol, Fortran and PL/1, and possibly other more specialized languages such as list-processing languages.

Since the conversion initiated by any particular format item is affected by the internal representation, standardization of the format language alone will not bring about complete uniformity of results. For example, the decimal representation of a fraction may terminate while its binary representation does not. Output of such a value, using a

format language which truncates, will yield a result which may depend upon the internal representation.

One way of standardizing a language is to designate a particular implementation of that language as the standard. This ensures that the consequence of any possible input of instructions or data is defined. By contrast, definitions of programming languages frequently define only the consequences of "valid" or "permissible" input.* A more thorough discussion of the problem is found in Lee (1968).

A combination of the UFL and DATA models, with the extensions described in Chapter IV, could serve as the basis for a "standard implementation" of a number of format languages. Its advantages over a conventional implementation are:

1. It employs conversational mode, returning immediate results of any trial.
2. The model is not restricted in any way by the characteristics of either the "host" computer or the external storage device, since both are represented by matrices. Therefore byte length, word length,

* For example, "Fortran vs. Basic Fortran", (Anon (1964)) contains the following passage:

"1.2 Scope

. . .

This specification does not prescribe:

. . .

(4) The results when the rules for interpretation fail to establish an interpretation of such a program."

etc. can be chosen with complete freedom.

5.5 DATA as a Teaching Aid

The DATA model could be useful in teaching of:

1. Data representations in a computer memory.
2. The organization of memory, and meanings of terms such as byte, field and record.
3. Input and output, item-oriented and record-oriented.

It could be used, for example, to illustrate these concepts as part of a course in any programming language. DATA, combined with UFL, could also be used to introduce students to format languages, although these are comparatively simple to learn.

The arguments in favor of DATA as a teaching aid are substantially the same as the arguments for introducing it in the first place. The conversational mode will encourage the student to experiment, even to the point of modifying his "copy" of the model itself. For example, if he wishes to change the byte length or the collating sequence and observe the results, he can do so without destroying the model. To return to the original version, he need only re-load it into his workspace. Furthermore, the model allows him to bypass many technical details, which can be especially troublesome on I/O. A beginning programmer may find himself in the position of not knowing whether his control cards, program,

data, or all three are in error. DATA allows him to dispense with control "cards" altogether, and to monitor the progress of an algorithm as closely as necessary. For example, a field or a record might be displayed after each step to ensure that the desired changes are occurring.

BIBLIOGRAPHY

- Anon, "FORTRAN vs. Basic FORTRAN - A Programming Language for Information Processing on Automatic Data Processing Systems", Comm. ACM 7, 10 (October 1964), 591-625.
- Bartee, T.C., Lebow, I.L., Reed, I.S., Theory and Design of Digital Machines, McGraw-Hill Book Co., New York, 1962.
- Breed, L.M. and Lathwell, R.H., "The Implementation of APL\360", Proceedings of the ACM Symposium on Interactive Systems for Experimental Applied Mathematics, 1967.
- Buchholz, W., (Ed.), Planning a Computer System, McGraw-Hill Book Co., New York, 1962.
- Cody, W.J., "The Influence of Machine Design on Numerical Algorithms", Proc. AFIPS 1967 Spring Joint Computer Conference, Thompson Books, Washington, D.C., 305-309.
- Conway, M.E., "Design of a Separable Transition-Diagram Compiler", Comm. ACM 6, 7 (July 1963), 396-408.
- Falkoff, A.D. and Iverson, K.E., "The APL\360 Terminal System", Proceedings of the ACM Symposium on Interactive Systems for Experimental Applied Mathematics, 1967.
- Flores, I., Computer Programming, Prentice-Hall, Englewood Cliffs, N.J., 1966.

- Halpern, M.I., "Toward a General Processor for Programming Languages", Comm. ACM 11, 1 (January 1968), 15-25.
- Hassitt, A., Computer Programming and Computing Systems, Academic Press Inc., New York, 1967.
- Hellerman, H., Digital Computer System Principles, McGraw-Hill Book Co., New York, 1967.
- IBM Corp., IBM System/360 Operating System PL/1(F) Programmer's Guide, Form C28-6594-1, 1966.
- IBM Corp., IBM System/360 Operating System PL/1 Language Specifications, Form C28-6571-4, 1965.
- IBM Corp., IBM System/360 Fortran IV Language, Form C28-6515-4, 1966.
- IBM Corp., IBM System/360 Operating System Cobol Language, Form C28-6516-5, 1965.
- Iverson, K.E., A Programming Language, John Wiley and Sons, Inc., New York, 1962.
- Iverson, K.E., "A Method of Syntax Specification", Comm. ACM 7, 10 (October 1964), 588-589.
- Knuth, D.E., et al, "A Proposal for Input-Output Conventions in Algol 60", Comm. ACM 7, 5 (May 1964), 273-283.
- Lee, J.A.N., "Current Research Toward the Standardization and Formal Definition of PL/1", Publication No. TN/CS/00001, University of Massachusetts, Amherst, Massachusetts.

- Lehmer, D.H., "Representation and Processing of Digital Information", Applied Combinatorial Mathematics, E.F. Beckenbach (Ed.), John Wiley and Sons, Inc., 1964, 6-11.
- Matula, D.W., "In-and-Out Conversions", Comm. ACM 11, 1 (January 1968), 47-51.
- McIntosh, T.M., "Syntax-Directed Analysis", unpublished M.Sc. thesis, University of Alberta, 1968.
- Mealy, G.H., "Another Look at Data", Proc. AFIPS 1967 Fall Joint Computer Conference, Thompson Books, Washington, D.C., 525-534.
- Pakin, S., *APL\360 Reference Manual*, to be published by Science Research Associates Inc., Chicago, 1970.
- Perlis, A.J., "A Format Language", Comm. ACM 7, 2 (February 1964), 89-97.
- Perlis, A.J., "The Synthesis of Algorithmic Systems", Jour. ACM 14, 1 (January 1967), 1-9.
- Standish, T.A., "A Data Definition Facility for Programming Languages", unpublished doctoral dissertation, Carnegie Institute of Technology, May 1967.
- Wegner, P., Programming Languages, Information Structures and Machine Organization, McGraw-Hill Book Co., New York, 1968.

APPENDIX A

A NUMERIC DATA IN MEMORY

A.1 Numbers and Their Representations

A positional system of representing numbers is one in which the contribution of any digit depends not only upon the digit itself, but also upon its position within the string. For example, in the evaluation of the decimal representation "223", the values of the digits in the hundreds, tens and ones positions are multiplied by "weights" of one hundred, ten and one, respectively. By contrast, in the evaluation of the representation "XCI" in Roman numerals (a non-positional system), the "X" is given a weight of -1 because of its position with respect to the "C", but its position with respect to the string as a whole has no effect upon its contribution. (Four positional representation systems are described by Lehmer (1964). One of these, the Chinese or modular system, has the potentially useful property that truly parallel arithmetic can be performed upon it because no carries or borrows are propagated.)

The discussion to follow will be restricted to the almost universal polynomial representation, also called the Arabic numeral system. Any instance of a polynomial representation employs a radix vector, of which the I th component is defined as the number of distinct digits any one of which may appear in the I th digit position of the representation. For example, since there are twelve inches in a foot and three feet in a yard, the inches and feet digit positions of the representation

of a length have radices of 12 and 3, respectively. This is an example of a "mixed radix" system. We shall assume all components of any radix vector to be equal, unless otherwise indicated. A symbol called the radix point may be used to separate the fractional and integer parts of a representation. If the radix point is omitted, its assumed position is to the right of the rightmost digit, giving the representation an integer value. Where the radix is ten, the radix point is called the decimal point.

The process whereby one representation of a number is replaced by another representation of the same number in a different radix is called a radix conversion. It is a transformation of the syntax which, ideally, has no semantic effect. However, an error may be introduced if the number contains a fraction which does not terminate quickly enough when represented in the new radix. The following table illustrates the fact that the radix used in representing a fraction determines how quickly the representation terminates.

	Radix	10	8
<u>Number</u>			
$1 \div 8$.1250	.1000
$2 \div 5$.400	.31463..

Any fraction which terminates after N fractional digits when represented in radix R_1 also terminates, after at most $K \times N$ fractional digits, when represented in radix R_2 , and vice versa, if there exists a positive integer K such that

$$\dagger R_1 = R_2 * K$$

The condition is sufficient but not necessary. For example, the fraction $1/2$ terminates after one fractional digit whenever the radix is even.

In a digital computer, data is represented by the discrete states of a number of bistable storage cells, each of which contains one bit of information. A storage unit of greater capacity, the register, consists of a logical grouping of cells. A register which is to represent an N -symbol alphabet must have at least $2 * N$ bits of information capacity, that is, it must contain at least $\lceil 2 * N \rceil$ storage cells. Where the number of cells in a register exceeds the required information capacity (for example, where $2 * N$ is not an integer), the probabilities of one or more possible register values are reduced to zero, with a corresponding loss of information capacity.

An example of an alphabet which must be encoded in a computer memory is the set of decimal digits. Each digit has an information content (assuming equal probabilities) of

2×10 , or approximately 3.322 bits. To be capable of encoding any decimal digit, a register must contain $\lceil 3.322 \rceil$ or 4 storage cells. Since 6 of the 16 possible values of such a register are unused, the efficiency of utilization of storage is $100 \times 3.322 \div 16$, or approximately 83%. In other words, the information content per storage cell averages only .83 bit. This system of encoding each digit of a decimal representation in a separate register is called binary-coded decimal, abbreviated BCD. The term is popularly used, synonymously with "alphanumeric", to include the encoding of alphabetic characters as well as digits. Where the alphabet consists of letters as well as digits, a minimum of $\lceil 2 \times 36 \rceil$, or 6 storage cells are required for each symbol. The extra $(2 \times 6) - 36$ characters are assigned to punctuation and special symbols.

Although the binary fixed-point and floating-point representations described in the following sections make more efficient use of storage cells than BCD does, this saving is outweighed in some applications by the time required for decimal-to-binary and binary-to-decimal conversions.

Computational problems may be grouped into two general classes, those where every digit is considered to be significant, and those where the number of significant digits is limited and independent of the magnitude of the number represented. Accounting problems are typical of the first class.

Every amount must be correct to the nearest cent, and is, in effect, an integral number of cents. The largest amount which can be represented is therefore limited by the number of digit positions in the representation.

Problems involving the results of physical measurements belong to the second class. A measurement is never exact, but the maximum relative error is likely to be approximately the same for each of a group of related measurements. The systems of representation which have developed for these two classes of problems, the fixed-point and the floating-point systems respectively, are discussed in the following sections.

A.2 Fixed-Point Representations

The previous section assumes that the reader is already familiar with the fixed-point notation and is aware, for example, that the order of increasing weights is from right to left.* Otherwise, terms such as "hundreds position" and "tens position" would be meaningless. This section discusses the fixed-point notation in more detail.

If we designate by D the vector of digits representing an unsigned integer in radix R , the polynomial by which its value is computed is

* The convention that weights increase from right to left is inconsistent with the practice of reading from left to right. Possibly it originated with the Arabs, from whom the system takes its name. (Lehmer (1964)).

$$+ / D \times R * (\rho D) - 1 \rho D .$$

The largest integer which can be represented is

$$-1 + R * \rho D ,$$

which occurs when

$$\pm \wedge / D = R - 1 .$$

For the more general case where there are N fractional digits (and $\pm N \geq 0$), the contribution of each digit is divided by $R * N$. The polynomial becomes

$$+ / D \times R * (\rho D) - N + 1 \rho D$$

and the largest representable number becomes

$$(-1 + R * \rho D) \div R * N$$

We have not yet considered the representation of negative numbers. Since the sign of a number is two-valued, we shall assume that the leftmost digit has a radix of two, with a 1 indicating minus and a 0 indicating plus. The range of integer values will now be

$$1-R*N-1 \quad \text{to} \quad -1+R*N-1 ,$$

including +0 and -0 as distinct values. The general expression for the value of a register becomes

$$(1-2 \times D[1]) \times +/D[1+i^{-1}+pD] \times R*N - 1 + i^{-1} + pD .$$

Another common method of representing negative numbers is to use one of the two "complements" of the number:

The radix-minus-one complement of the number C , represented (with radix of R) as $C_1C_2C_3C_4 \dots C_N$ is defined as $(-1+R*N)-C$. It is formed by subtracting each digit from $R-1$.

The true complement of the number C is defined as $(R*N)-C$. It is formed by adding 1 to the radix-minus-one complement.

The sign digit is treated as a binary digit, regardless of the radix used. For example, when the other digits are subtracted from $R-1$ in forming the complement, the sign digit is subtracted from 1. When an addition is to be performed, negative numbers are complemented (if they are not already in complement form), the addition is performed, and the result, if negative is complemented (unless it is to be stored in complement form). Subtraction consists of complementing and adding. This is the advantage of the complement notation - the expense of a subtraction circuit is avoided.

If a carry is generated from the sign digit, it is treated as follows: in a radix-minus-one system, add one to C_N . In a true complement system, drop the carry digit.

In the design of a computer which is capable of performing arithmetic upon binary-coded decimal operands, one of the factors affecting the choice of a code to represent the decimal digits is the question of how easily the appropriate complement can be formed. Since any complemented negative number could also be interpreted as some uncomplemented positive number, the sign bit is still required.

A.3 Floating-Point Representations

Scientific notation is frequently used to express numbers for which the fixed-point representation has insufficient range, or would require an inconvenient number of leading or trailing zeros. The floating-point representation, an adaptation of scientific notation, serves the same purpose in a computer memory. The number to be represented is converted to the form

$$F \times R^E, \text{ where } E \text{ is integer.}$$

We shall assume a normalized representation, with a fractional value of F , i.e.,

$$\frac{1}{2} \leq F < 1, R^E \geq 1, R \neq 1.$$

Since R , the radix of the representation, need not be represented as part of the data item, the floating-point representation consists of the signed fixed-point representations of E and F . The range of E , and therefore the range of the entire representation, is restricted by the number of digit positions available for the representation of E . If we designate by $EMAX$ the largest possible absolute value of E , then any number A such that

$$\pm (|A|) \geq R * EMAX$$

is too large to be represented with fractional F , and is, in effect, infinite. Similarly, a number which is too small in absolute value to be expressed in the form

$$F * R * E \quad \text{where} \quad \pm E = -EMAX$$

$$\pm F > 0 ,$$

is indistinguishable from zero and is called an infinitesimal. Clearly, infinitesimals and infinite "numbers" exist for any floating-point system for which $EMAX$ is finite. Also, if we designate by N the number of digits in the representation of F , then the smallest detectable increment in the value of $F * R * E$ (resulting from a change of 1 in the least significant digit of F) is

$$(RF^{*-N}) \times R * E ,$$

where RF is the radix used in representing F . The unavoidable error in the representation is bounded by

$$.5 \times (RF^{*-N}) \times R * E ,$$

which approaches zero as N increases.

Since any floating-point (or fixed-point) representation consists of an integer with a scale factor, only rational numbers can be represented without error. Where N is finite, the range of representable numbers is restricted further, to some subset of the rational numbers. Consequently, the results of computer arithmetic do not always conform to mathematical results. Cody's attempts to find the identity element under floating-point multiplication illustrate this anomaly. (Cody (1967)).

APPENDIX B

DATA Programmer's Guide

CHAPTER B1

INTRODUCTION

This Programmer's Guide describes the DATA language in detail. The reader is assumed to be familiar with APL, though not necessarily proficient in it. DATA appears to the user as an implementation of a simple conversational programming language. Physically, it resides entirely within one APL workspace.

B.1.1 Definitions of Terms

Data Item - A single number or non-numeric "value".

Memory - A storage area within the computer, accessible by the program.

Data Set - A collection of data, outside of memory, (i.e., not directly accessible by the program). There are three data sets, identified by the integers 1, 2 and 3.

Field - The portion of memory, or of a data set, which encodes a single data item. The portion of memory to which an identifier refers.

Picture - An expression which specifies the representation to be assigned to a particular field.

Record - A group of fields. A single record-oriented I/O statement causes one record to be made accessible to the program (input), or to be designated for transmission to a data set (output).

I/O - Input or output or both. Any transfer of data between memory and a data set.

Byte - A single storage location, either in memory or in a data set. A byte contains 6 bits.

Character - The contents of any byte. Any one of the 6^4 possible combinations of 6 binary digits (bits).

Character Set - The ordered set of graphic symbols represented by characters. The DATA character set consists of the vector *DATASEQ*, where

$$\dagger \wedge / \text{DATASEQ} = ' \text{ABCDEFGHIJKLMNOPQRSTUVWXYZ}\alpha?,.$$

$$0123456789()[]+-\times\div=/'',12\rho'\circ'$$

For the character which, interpreted as an unsigned binary integer, yields the value N , the corresponding symbol is *DATASEQ*[$N+1$]. The " \circ " entries in *DATASEQ* indicate characters to which no symbol has been assigned.

B.1.2 Memory Organization

The memory contains several record-areas, each of which can accommodate a single record of any length up to a maximum of 63 characters. These record-areas are referenced by numbers. Execution of the statement "*DECLARE N*" (see Chapter B2: Declarative Statements) ensures that at least $N+1$ record-areas exist, numbered $0, 1, \dots, N$. Although there is no specific limit to either the number of record-areas which can be created, or the number of records which can be written into a data set, the total storage available for memory, data sets and computations is limited by the size of the APL workspace.

B.1.3 Data Representations

Any field contains a sequence of bits. The interpretation of those bits depends upon the picture item associated with the field. The four systems of representing data are:

Binary Fixed-Point - The corresponding picture item contains Z's or 9's or both; a V is optional; a + or a - is optional; R, X and . are not permitted.

A binary fixed-point field can represent any number which can be represented by replacing all Z's and 9's of the picture item by decimal digits, replacing the V (if any) by a decimal point, and replacing the + or - by a sign. If there

is no decimal point, the value is an integer. If there is no sign, it is positive or zero.

Table B.2 of the appendix to this Programmer's Guide shows the field lengths required for any number of decimal digits. The table is always correct, and is adequate for the majority of programming problems. However, a more precise statement follows for the benefit of more experienced programmers:

Where the field length is N bytes, and S has the value 1 if there is a sign and 0 otherwise, the largest absolute value which the field can represent is 10^{N-S} , assuming there is no V in the picture item. If there is a V , with M digit positions to the right of it, the above expression must be divided by 10^M .

For example, where the picture item is "9", the integers 0 to 63 can be represented. Where the picture item is "+9", the range is -31 to +31.

Binary Floating-Point - The corresponding picture item contains Z's or 9's or both; one . or one X or both; a + or a - is optional; R and V are not permitted. (X may be read as "times 10 to the power".) If we disregard all characters to the right of the X , if any, the number of remaining digit positions (Z or 9) in the picture item is the number of significant figures required.

A binary floating-point field can represent any number with absolute value between $8 * 10^{-64}$ and $8 * 10^{63}$, or zero. (That is, between $2.6 * 10^{-52}$ and $6.0 * 10^{50}$.) Either 5 or 9 significant figures are carried, depending upon the picture item.

Alphanumeric - The corresponding picture item contains one or more α 's. Each α represents a member of the character set (defined in a previous section), encoded in one byte.

Binary-Coded Decimal (BCD) - The corresponding picture item contains Z's or 9's or both, and an R . (The R suggests "report", since BCD fields can be printed without conversion.)

A BCD field is similar to an alphanumeric, except that the arrangement of characters (decimal digits, decimal point, sign, etc.) must form a valid decimal representation.

B.1.4 Reserved Words

The following identifiers, plus the function names listed in Appendix C, part 1, are reserved words which may not be redefined by the programmer.

BUF

CHARSET

CORE

DATASEQ

DATASET

END

ENDFILE

IDENT

INDEX

ITEMIN

ITEMOUT

OUT

OUTPTR

PICT

RANK

RECL

TABLE

TALLY

CHAPTER B2

DECLARATIVE STATEMENTS

A declarative statement is used to subdivide one record-area of memory into fields, and to assign an identifier and a picture (i.e., a description of an internal representation) to each field. The declarative statement is initiated by input of either (a) *DECLARE INITIAL* integer or (b) *DECLARE* integer, where "integer" is the record-area number. Statement (a) will cause deletion of all previously-defined fields in that record-area. Statement (b) will allow additional fields to be defined, without affecting those which are defined already.

In response to input of statement (a) or (b), the keyboard will unlock to accept input of an identifier. The procedure for creation of fields is as follows:

1. When the keyboard unlocks, (if more identifiers are to be defined in this record-area) type the desired one-character identifier, followed by a "." if it is to reference an array. If no more identifiers are to be defined, exit by typing an immediate carriage return.

If an array is being defined, proceed to step (2).

Otherwise, to to step (3).

2. When "□:" appears, type the rank vector of the array, of not more than three elements. (An empty rank vector will nullify the effect of the "°", and yield a scalar.)
3. When the keyboard unlocks, type the picture specification of the desired internal representation of the field (a single element, if an array). The picture language is defined in the appendix to this Programmer's Guide.
4. If the argument of the initiating statement was 0, to to (1).

Otherwise, when "□:" appears, type: the number designating the position, within the record, of the first character of this field; or a vector of integers specifying the positions of all characters of this field; or a list, enclosed in quotes, of identifiers already defined in this record-area which are to be included hierarchically under this identifier.

5. Go to (1).

In record-area 0, the user cannot specify the position of a field (step (4), above), and is thereby freed from responsibility for computing the length in memory of each field he defines. This record-area is provided to allow definition of fields which are not inherently part of any record. Since the identifiers referencing these fields never

require qualification, they are valid operands for *GET* and *PUT* statements. (See the chapter on Assignment Statements for a discussion of qualified identifiers.)

Execution of the statement "*CLEARΔPICTURES*" deletes all fields which have been defined in any record-area.

Execution of the statement "*SHOWPICTURES* integer", where "integer" is a record-area number, causes a table to be printed showing all the identifiers defined in that record-area, with the picture item, field length and position, and rank vector (if any) for each.

CHAPTER B3

ASSIGNMENT STATEMENTS

Any assignment statement of the DATA language contains one or more of the words:

MOVETO

MOVE△BY△NAME△TO

ASSIGN△TO

CONTENTSOF

BLANK△RECORD

OF

SUB

The purpose of an assignment statement is to retrieve the contents of one or more fields in memory, or to assign new contents to those fields. The effect of each form of assignment statement is described in the following paragraphs.

identifier-1 *MOVETO* identifier-2 - The contents of the source field, referenced by identifier-1, are moved into the receiving field, referenced by identifier-2. The contents of the source field are left-justified in the receiving field, and

truncated or filled with binary zeros* if the lengths are different. The picture items for the two fields are not checked. If they are different (especially in length), unexpected results are likely to occur.

`record-area-1 MOVE△BY△NAME△TO record-area-2` - This statement is equivalent to a series of `MOVETO` statements, one for each identifier which is defined in both record-areas. For example, if identifiers *A*, *B* and *C* are defined in record-area-1 and *A*, *C*, *D* and *E* are defined in record-area-2, only the fields referenced by *A* and *C* will be moved.

`(APL expression) ASSIGN△TO identifier` - The result of evaluation of the APL expression is encoded in the form specified by the picture item corresponding to the identifier, and stored in the field reference by it. The result of the expression must be a numeric scalar if the picture item is numeric, or a character scalar or vector if the picture item is alphanumeric. .

* Binary zeros have the effect of setting binary numeric fields to zero, and filling alphanumeric and BCD fields with blanks.

In the latter case, the vector will be left-justified in the field and truncated or blank-filled on the right if the length of the vector is not equal to the length of the field.

X←CONTENTSOF identifier - The APL variable *X* will assume the value of the numeric scalar, or character scalar or vector referenced by the identifier. Since "*CONTENTSOF* identifier" is an APL expression, it may appear as the left-hand argument of an *ASSIGNΔTO* statement (above). For example,

(*CONTENTS OF 'B'*) *ASSIGNΔTO 'A'*

is similar to *B MOVETO A* except that in the former case the picture items associated with both identifiers are used to perform a data conversion.

BLANKΔRECORD integer - Record-area number "integer" will be filled with binary zeros, that is, binary numeric fields will be set to zero and BCD and alphanumeric fields will be blanked.

OF - Is used to qualify an identifier. That is, where the same identifier is defined in two or more record-areas, a reference to it may be qualified

by the word "OF" followed by the record-area number. For example, 'A' OF 2 and 'A' OF 1 reference two different fields. (A qualified identifier often must be parenthesized because of the right-to-left rule of APL.) An unqualified identifier is assumed to refer to the lowest-numbered record-area for which that identifier is defined. As a result, identifiers in record-area 0 never require qualification. All the identifiers referred to in the above paragraphs may be qualified or unqualified.

SUB - Is used to index an identifier. Any identifier which references an alphanumeric array may appear with or without indices in an assignment statement. The identifier without indices references the character vector consisting of the array elements in row-major order. Each element of an array M is a character scalar or vector (depending upon the picture item). Any one element may be referenced by the identifier and $\rho\rho M$ indices. For example, where M is defined as a 3×2 matrix (in record-area 0), described by picture item "[2]α", the following statements might be executed:


```
'ABCDEFGHijkl' ASSIGNΔTO 'M'
```

```
□ ← CONTENTSOF 'M' SUB 3 2
```

```
KL
```

```
□ ← CONTENTS OF 'M'
```

```
ABCDEFGHijkl
```

```
□ ← CONTENTSOF 'M' SUB 1 2
```

```
CD
```

A qualified identifier may be indexed as well. If *M* had been defined in record-area 2, these statements might be executed:

```
'PQ' ASSIGNΔTO ('M' OF 2) SUB 2 2
```

```
□ ← CONTENTSOF 'M' OF 2
```

```
ABCDEFpqijkl
```

Although numeric arrays may be defined, the present version of DATA requires every appearance of an identifier referencing such an array to be indexed. That is, only a single element may be referenced at one time, and the advantage of the array is lost. The use of numeric arrays is therefore not recommended and will not be discussed further.

CHAPTER B4

INPUT AND OUTPUT STATEMENTS

Transmission of data from a data set to memory is called input, and the inverse operation is called output. The abbreviation I/O refers to input or output, or both. Two forms of I/O are provided, called item-oriented and record-oriented.

The item-oriented input and output statements are, respectively, *GET* list and *PUT* list, where "list" is a character vector consisting of one or more unqualified, unsubscripted identifiers. (Record-area 0 is provided for this purpose, although other record-areas may also be used. See the chapter on Assignment Statements.) The order in which identifiers appear in the vector corresponds to the order in which fields are arranged on the data set. On input, the N^{th} field retrieved from a data set by a particular *GET* statement is transmitted to the field referenced by the N^{th} identifier in the argument of that statement. On output, the N^{th} identifier determines (by means of its picture item and the contents of the field referenced by it) the contents of the N^{th} field transmitted to the data set.

A statement of the form "*ITEMIN+N*" (where $\pm N \in 13$) must be executed to specify which data set is to be accessible to item-oriented input statements. A field in that data set consists of a group of non-blank characters surrounded by

blanks. The representation employed must be alphanumeric or BCD. Fields must be separated by blanks (except that the end of a record may serve as the closing delimiter of a field), and must contain no imbedded blanks. Such a data set may be created by a program, or by execution of a sequence of *STACK* statements (see Chapter B5).

A programmer who is familiar with the binary representations used in DATA can, in some cases, use a *GET* statement to access a data set which contains binary fixed-point or floating-point fields. However, since the binary representation of zero is identical with the alphanumeric representation of a blank, the current value of any binary field would affect the "length" of that field and, therefore, the correctness of the results.

A statement of the form *ITEMOUT+N* must be executed to specify the output data set. The picture item associated with each identifier determines the representation to which the contents of the field referenced by that identifier are to be converted. (See the Chapter on Declarative Statements for a description of the picture language, used as an output format language.) Blanks are added to the left and right of each field (before transmission to the data set) for readability if the data set is to be printed, and to serve as delimiters if it is later used for item-oriented input. In the latter case, imbedded blanks within any field must be avoided.

The fields specified for output are not transmitted one by one, but are collected and grouped in a portion of memory called a buffer. The groups of fields, called records, are transmitted to the data set. After execution of the final *PUT* statement, the statement *CLOSEΔFILES* must be executed to transmit as a final record the fields which remain in the buffer. The statement *PRINT ITEMOUT* may then be executed to cause the contents of the data set created by item-oriented output to be displayed at the terminal. (See Chapter B5, Control Statements.)

Note that there is no connection between the grouping of fields into records and the grouping of identifiers into *GET* and *PUT* statements. For example, the sequence of statements

```
verb 'A'  
verb 'BCD'  
verb 'EF'
```

is exactly equivalent to the single statement

```
verb 'ABCDEF'
```

where "verb" is either *GET* or *PUT*.

A record-oriented I/O operation transfers a single record from a data set to memory, or vice versa.. Identifiers and

picture items do not have any effect, since no conversion is performed. While the record resides in memory, the programmer can use an identifier to refer to a field in order to retrieve or modify its contents. The picture item associated with that identifier determines the conversion (if any) which is required. These identifiers and picture items are defined by declarative statements (see Chapter B2).

Record-oriented I/O statements are of the form *N* verb *K*, where *N* is the data set number, *K* is the record-area number, and "verb" is *READΔINTO* or *WRITEΔFROM*.

Each record-area in memory is large enough to accommodate a record of maximum length. Where the length of a record which is read from a data set is less than the maximum, the remaining characters in the record-area are left unchanged.

On output, the programmer has control over the length of each record which is transmitted. Regardless of the lengths and positions of fields which have been declared, any record will be just long enough to include the rightmost field to which a value has been assigned since the last execution of a *WRITEΔFROM* statement referring to that record-area. For example, execution of the sequence of statements:

```
(APL expression) ASSIGNΔTO 'X' OF K
N WRITEΔFROM K
N WRITEΔFROM K
```


will cause only one record (containing at least the one field referenced by 'X') to be written into the data set. The second *WRITEΔFROM* statement will have no effect, because no new data has been assigned.

CHAPTER B5

CONTROL STATEMENTS

The statements of the DATA language which contain the following verbs will be called control statements:

REWIND

PRINT

STACK

CLEARΔTAPES

ENDFILE

They are used as follows:

REWIND N - where $1 \leq N \leq 3$, is used to reposition one or more data sets to their initial records. After a data set has been created by a sequence of output statements, it must be rewound in order for subsequent input statements to access it.

PRINT N - causes the contents of data set *N* to be displayed at the terminal, one record per printed line. The contents of the data set will be assumed to be either BCD or alphanumeric. The printed representations of binary fixed-point or floating-point fields, if any, will

be unintelligible.

The entire data set will be printed. It is not necessary to rewind either before or after printing.

N STACK string - where *N* is the data set number, and "string" is a character vector each element of which is a valid DATA character (i.e., a member of the character set *DATASEQ*. See Chapter B1.) The *STACK* statement is used to prepare a BCD and/or alphanumeric data set for input. Since each execution of a *STACK* statement adds records to the data set, it must be rewound before the first statement and again after the last. The "string" may contain any number of characters up to and including 62.

CLEARΔTAPES - This statement has the effect of erasing and rewinding all three data sets.

→*ENDFILE[N]/ENDJOB* - any input statement, whether record-oriented or item-oriented, may encounter an end-of-file condition, that is, there may not be sufficient data in the data set to satisfy the input statements which are executed. The results of this condition are as follows:

Record-Oriented - The record-area of memory will be unchanged, and the global variable *ENDFILE[N]* (where *N* is the data set number) will be set to 1 (its value is 0 otherwise).

Item-Oriented - *ENDFILE[ITEMIN]* will be set to 1, and an end-of-file message will be printed by the terminal. Where the argument of a *GET* statement contains more than one identifier, an end-of-file condition may occur after some but not all of the corresponding fields have been assigned new values. The remaining fields will then be left unchanged.

APPENDIX TO DATA PROGRAMMER'S GUIDE

AB.1 The DATA Picture Language

We shall refer to the set of interpretation rules for DATA picture items as a "picture language". When a value is being assigned to a field, the corresponding picture item indicates which of the four internal codes will be used. On output, the picture item serves as a format statement to describe a specific BCD representation.

AB.2 Syntax of the Picture Language

The rules of syntax are shown in Table B.1.

AB.3 Semantics of the Picture Language

AB.3.1 Usage

A picture item may specify a representation in memory (upon execution of an assignment statement), or in a data set (upon execution of an output statement). Interpretation is as follows:

1. Assignment - The picture item for any field is used by the input verb *GET* or the assignment verb *ASSIGNΔTO* to assign a value to that field. A picture item containing an "α" indicates alphanumeric data, one character per byte.

p	DIGIT POSITION	z 9
d	DIGIT	0 1 2 3 4 5 6 7 8 9
1	INTEGER	d d 1
2	REPLICATOR	[1]
3	INSERTION	. v , _ EMPTY
4	SIGN	+ - EMPTY
5	QUALIFIER	T R EMPTY
6	INTEGER FIELD	p 2 p p 6
7	NUMBER FIELD	6 6 3 7
8	NUMERIC	4 7 4 7 x 4 6
9	ALPHANUMERIC	α α 9 2 α
10	PICTURE ITEM	8 5 9

Table B.1 Rules of Syntax of DATA

Picture Language

A numeric picture item which does not contain an "R" indicates a binary fixed-point or binary floating-point internal representation. Since each "Z" or "9" marks a decimal digit position, the internal representation must be capable of carrying at least as many significant (decimal) digits as there are "Z's" and "9's" in the NUMBER FIELD part of picture item (see Table B.1). Table B.2 shows the field length corresponding to each number of decimal digits.

Where the picture item contains an "X" or a "." or both, and no "R", the internal representation will be binary floating-point. Where neither "X", "R" nor "." appear, binary fixed-point is indicated.

Where a picture item contains an "R" (standing for "report", to suggest a representation which can be printed), that BCD representation which would be created on output using the picture item as a format item becomes the internal representation. It occupies one byte for each character of the picture item other than *T*, *R* and *V*.

Binary floating-point representations are

Minimum
Field Length in Memory (Bytes)

Number of Decimal Digits	Fixed-Point		Floating -Point
	Signed	Unsigned	
1	1	1	4
2	2	2	4
3	2	2	4
4	3	3	4
5	3	3	4
6	4	4	7
7	5	4	7
8	5	5	7
9	6	5	7

Table B.2 Field Lengths of Fixed-Point and
Floating-Point Representations

always signed. A binary fixed-point representation is signed if the picture item contains a "+" or a "-", otherwise it is unsigned. Note that the absence of a sign may affect the field length (Table B.2).

2. Output - The picture item for any field which is to be output using a *PUT* statement determines the external BCD representation which will be created. Data items for which the picture item is alphanumeric or BCD are transmitted without conversion, that is the external representation is the same as the internal. Interpretation of a picture item to describe a BCD representation is as follows:

AB.3.2 Digit Positions

A "Z" indicates a digit position which will be blank if that digit is a leading zero, that is, if it is zero and there are no non-zero digits to the left of it. (A Z may not appear to the right of 9 in a picture item.) Thus, for example, a picture containing Z's but no 9's will convert the number zero to a blank field.

A "9" indicates a digit position which must contain a digit, never a blank.

AB.3.3 Insertions

_ indicates a blank character position.

X indicates the position of an X, which will be followed by an exponent field.

, indicates the position of a comma, which will be replaced by a blank if no digits appear to the left of it.

. indicates the position of a decimal point in the representation.

V indicates the location of the decimal point, for the purpose of scaling. However, no decimal point appears in the representation.

AB.3.4 Qualifiers

T indicates truncation. Absence of a T indicates that the result is to be rounded away from zero, that is, the absolute value is rounded up.

AB.3.5 Signs

+ indicates the position of a plus or a minus sign.

- indicates the position of a minus sign or a blank.

Where the picture item does not contain a sign, the absolute value of the data item will be transmitted.

APPENDIX C

Transition Diagrams and APL Functions

C.1 Format Languages

Figures C.1 and C.3 contain the transition diagrams which have been chosen to represent the format languages of Fortran and PL/1. An arc of a transition diagram may be identified by either

1. An integer, referring to a transition diagram.
2. One or more terminal symbols, any one of which may appear.

Otherwise, that arc represents a blank path.

Each transition diagram is identified by a number. Where multiple exits exist, the circle representing each exit node contains another number, which is used in the semantic analysis. Matrix representations of these transition diagrams, together with other arrays used by the UFL system, are also included in this appendix.

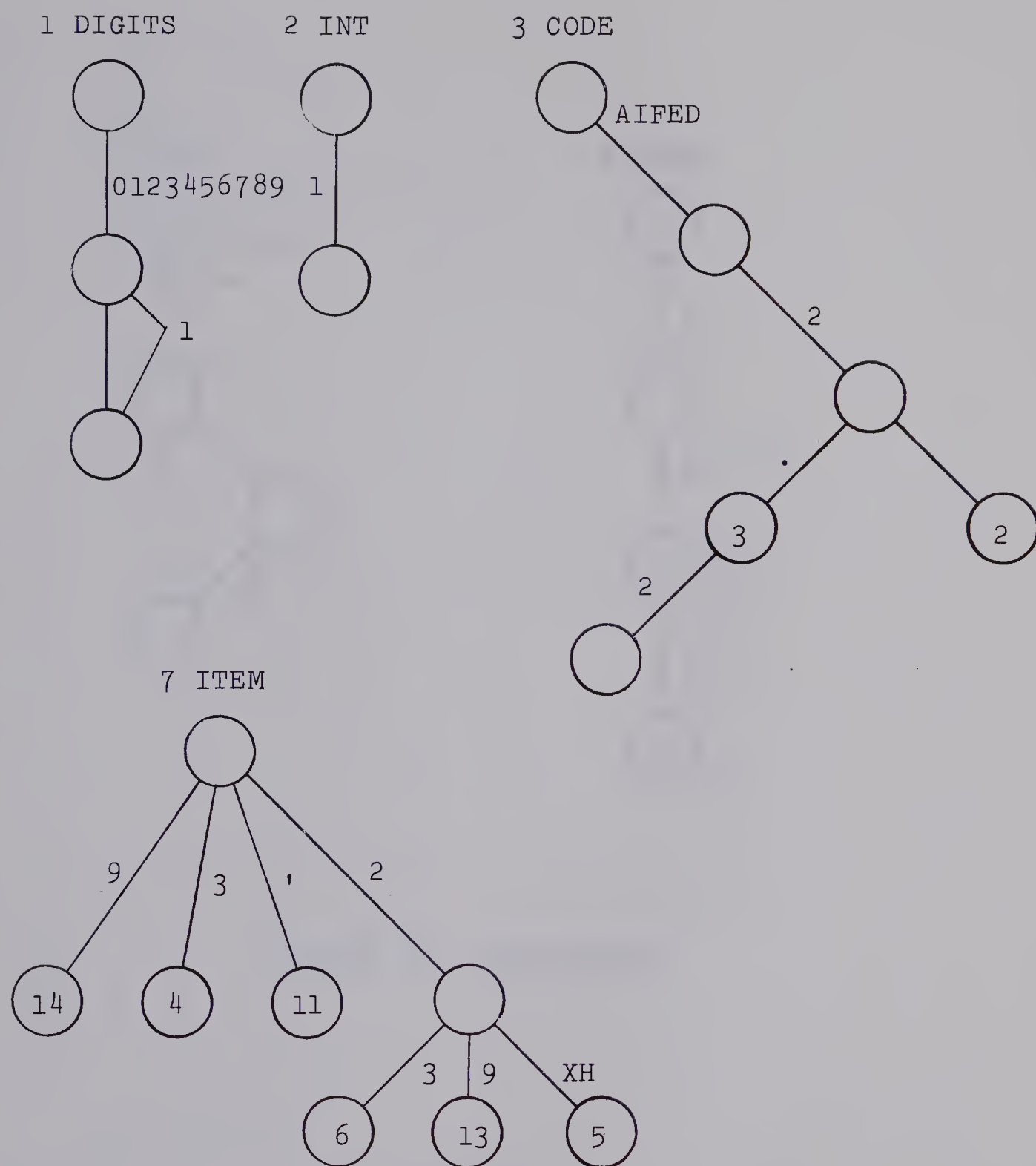
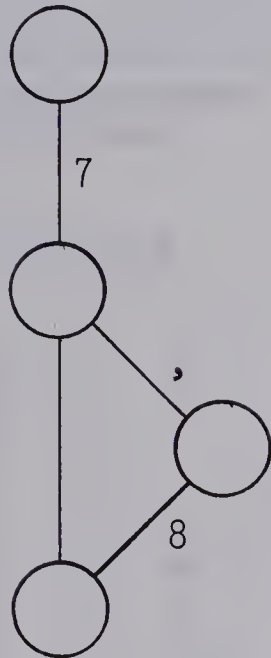


Figure C.1 Fortran Transition Diagrams

8 LIST



9 FORM

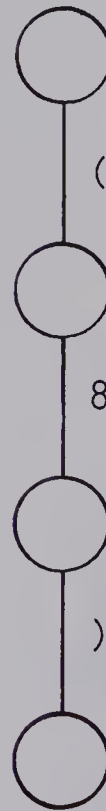


Figure C.1 Continued

▽ FORTFORM FMT

```
[1]  LANGCODE←1
[2]  'SAVE CODE?'
[3]  →SKIP 'Y'=1ρ□, ' '
[4]  FNEXT←1+MNEXT←0
[5]  FORM←FMT
[6]  TABLE←FORTTABLE
[7]  CHINDEX←FORTCHINDEX
[8]  CHARS←FORTCHARS
[9]  TYPES←FTYPES
[10] NEXTITEM
```

▽

FORTTABLE

1	1	2	0	22	0	0	
1	2	3	1	1	-1	0	FTYPES
1	2	3	0	0	-1	0	
3	1	2	0	20	0	10	
3	2	3	1	2	0	0	DIGIT
2	1	2	1	1	1	0	WIDTH
3	3	5	0	25	0	0	CODE
3	5	6	1	2	3	0	ITEM
3	3	4	0	0	2	0	LIT
7	1	2	1	2	0	0	RPITM
7	2	5	0	24	5	0	
7	2	3	1	3	6	0	LIST
7	2	3	1	9	13	0	FORM
7	1	4	1	3	4	0	
7	1	4	1	9	14	0	LITRL
7	1	5	0	27	11	0	
8	1	2	1	7	0	0	REPFM
8	2	3	0	21	0	0	
8	3	4	1	8	8	0	
8	2	4	0	0	8	0	
9	1	2	0	28	0	12	
9	2	3	1	8	0	0	
9	3	4	0	29	9	0	

FORTCHINDEX

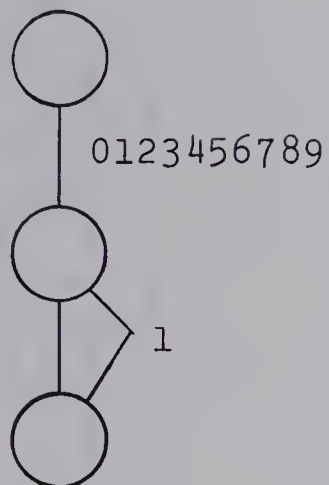
22	22	22	22	22	22	22	22	22	22	20	20
	20	20	20	24	20	20	20	20	24	20	25
	21	27	28	29	21	0					

FORTCHARS

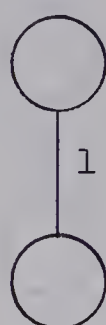
0123456789ADEFGLPTXZ.,'()/

Figure C.2 Fortran Arrays

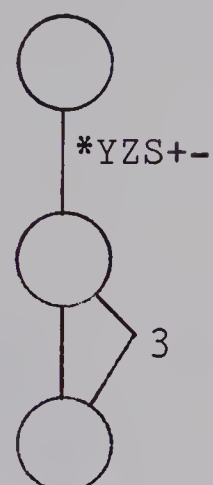
1 DIGIT



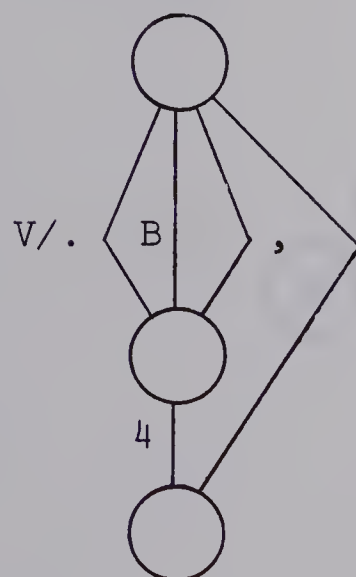
2 INT



3 ZEDS



4 SEP



5 GRP

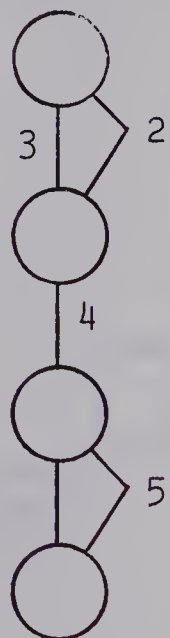
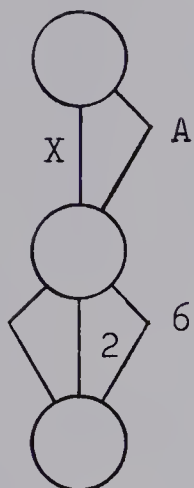
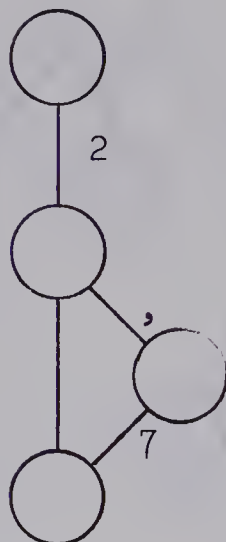


Figure C.3 PL/1 Transition Diagrams

6 PICT



7 SPEC



8 ITEM

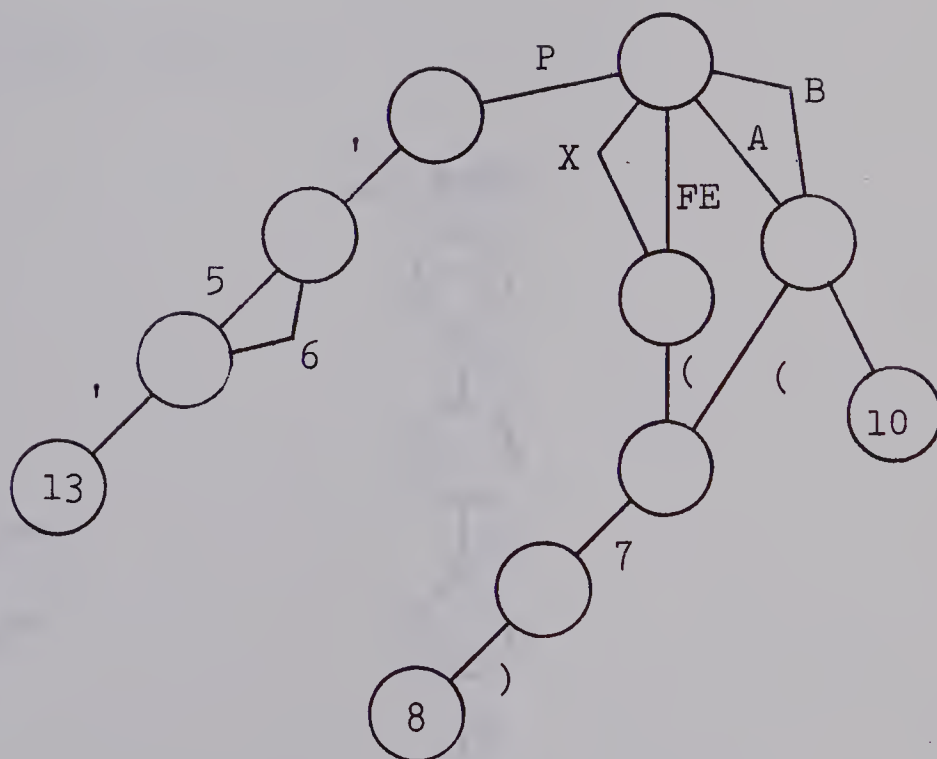


Figure C.3 Continued

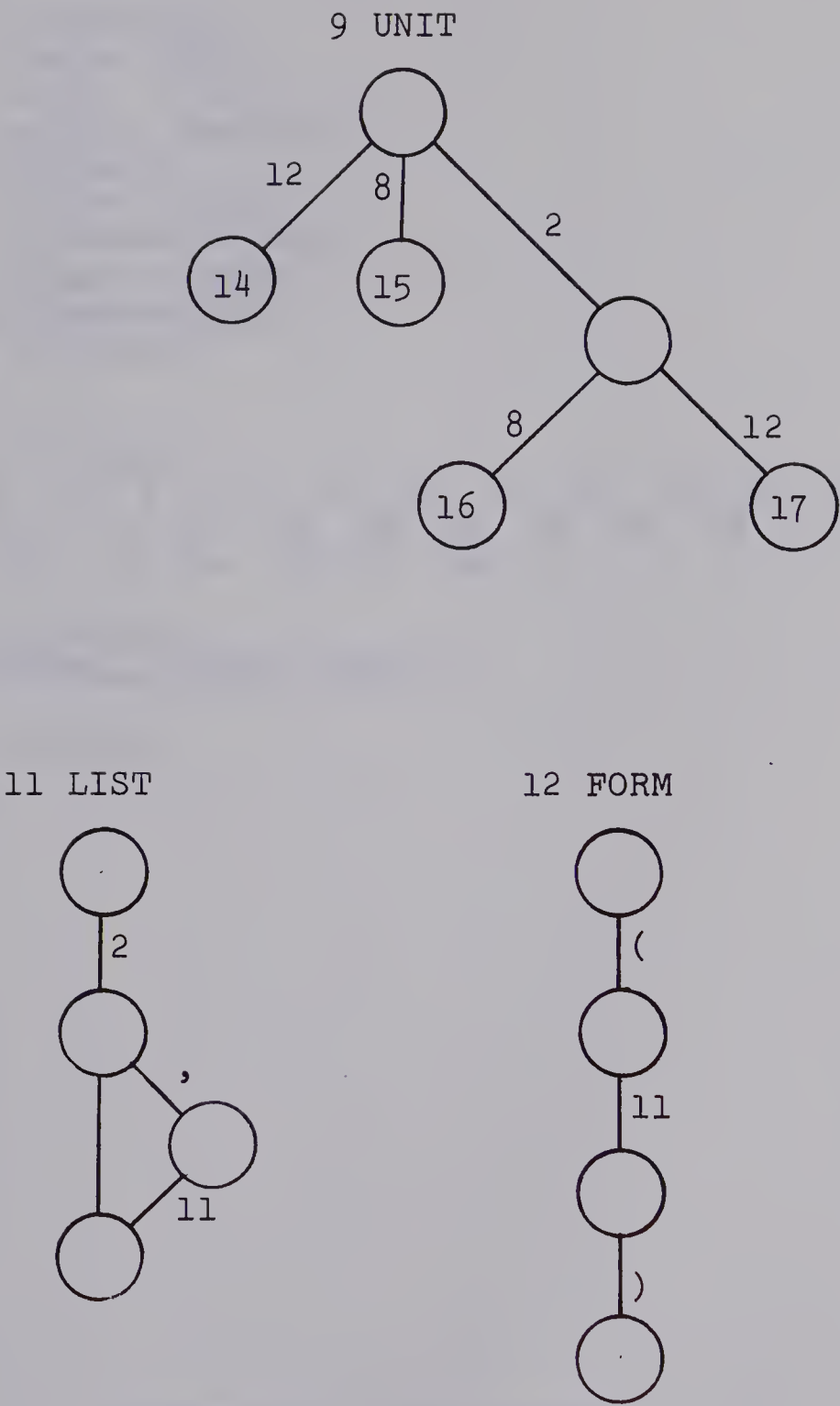


Figure C.3 Continued

▽ PLFORM FMT

```

[1]  LANGCODE←2
[2]  'SAVE CODE?'
[3]  →SKIP 'Y'=1ρ', ' - '
[4]  FNEXT←1+MNEXT←0
[5]  FORM←FMT
[6]  TABLE←PLTABLE
[7]  CHINDEX←PLCINDEX
[8]  CHARS←PLCHARS
[9]  TYPES←PLTYPES
[10] NEXTITEM

```

▽

PLCINDEX

22	22	22	22	22	22	22	22	22	22	22	21	25
	24	32	34	34	34	34	34	34	34	38	20	20
	19	24	28	29	27	25	0					

PLCHARS

```
0123456789,./B*YZS+-PFEAX()'
```

PLTYPES

```

DIGT
INT
ZEDS
SEP
GRP
PICT
SPEC
ITEM
UNIT

```

```

LIST
FORM

```

Figure C.4 PL/1 Arrays

1	1	2	0	22	0	0
1	2	3	1	1	-1	0
1	2	3	0	0	-1	0
2	1	2	1	1	2	0
3	1	2	0	34	0	0
3	2	3	1	3	3	0
3	2	3	0	0	3	0
4	1	2	0	25	0	0
4	1	2	0	21	0	0
4	1	2	0	32	0	0
4	1	3	0	0	4	0
4	2	3	1	4	4	0
5	1	2	1	3	0	0
5	1	2	1	2	0	0
5	2	3	1	4	0	0
5	3	4	1	5	5	0
5	3	4	0	0	5	0
6	1	2	0	24	0	0
6	1	2	0	19	0	0
6	2	3	1	2	6	0
6	2	3	1	6	6	0
6	2	3	0	0	6	0
7	1	2	1	2	0	0
7	2	3	0	21	0	19
7	3	4	1	7	7	0
7	2	4	0	0	7	0
8	1	2	0	38	0	0
8	2	3	0	27	0	0
8	3	4	1	5	0	0
8	3	4	1	6	0	0
8	4	5	0	27	13	0
8	1	6	0	19	0	9
8	1	6	0	32	0	9
8	6	8	0	28	0	0
8	6	11	0	0	10	0
8	1	7	0	20	0	9
8	1	7	0	24	0	9
8	7	8	0	28	0	0
8	8	9	1	7	0	0
8	9	10	0	29	8	0
9	1	2	1	2	0	0
9	1	3	1	12	14	0
9	1	4	1	8	15	0
9	2	5	1	8	16	0
9	2	6	1	12	17	0
11	1	2	1	9	0	0
11	2	3	0	21	0	0
11	3	4	1	11	11	0
11	2	4	0	0	11	0
12	1	2	0	28	0	18
12	2	3	1	11	0	0
12	3	4	0	29	12	0

Figure C.5 "PLTABLE"

C.2 Listing of Functions

C.2.1 The DATA Model

<u>Function Name</u>	<u>Page</u>	<u>Remark</u>
<i>ALPHAΔCODE</i>	133	Character-to-binary conversion
<i>ALPHAΔTOΔCHAR</i>	132	Binary-to-character conversion
<i>ASSIGNΔTO</i>	130	Assignment statement
<i>BFLΔAPL</i>	131	Floating-point to numeric
<i>BFXΔAPL</i>	131	Fixed-point to numeric
<i>BLANKΔRECORD</i>	130	Assignment statement
<i>CLEARΔPICTURES</i>	130	Declarative statement
<i>CLEARΔTAPES</i>	130	Control statement
<i>CLOSEΔFILES</i>	128	I/O statement
<i>CODEΔFRACT</i>	132	Called by <i>BFLΔAPL</i>
<i>CONTENTSOF</i>	128	Assignment statement
<i>DECLARE</i>	127	Declarative statement
<i>ELEMENT</i>	131	Numeric-to-character conversion
<i>EMPTYΔBUFFER</i>	135	Called by <i>PUT</i> , <i>CLOSEFILES</i>
<i>EPORTCODE</i>	133	BCD to binary conversion
<i>EXPANDK</i>	130	Expand any array
<i>EXTERNAL</i>	132	Code conversion
<i>EXΔFORMAT</i>	135	Expansion of items
<i>FIXΔCODE</i>	132	Numeric to fixed-point
<i>FLOATCODE</i>	133	Numeric to floating-point

<u>Function Name</u>	<u>Page</u>	<u>Remark</u>
<i>GET</i>	128	I/O statement
<i>GROUP</i>	130	Called by <i>SETPICTURE</i>
<i>INITIAL</i>	127	Declarative statement
<i>INTERNAL</i>	133	Code conversion
<i>LOADΔBUF</i>	136	Called by <i>GET</i>
<i>MOVETO</i>	127	Assignment statement
<i>MOVEΔBYΔNAMEΔTO</i>	128	Assignment statement
<i>NAMEΔTYPE</i>	136	Tests for qualifier
<i>NUMREP</i>	135	Character-to-numeric conversion
<i>NUMΔSIGΔBITS</i>	133	Code conversion
<i>OF</i>	129	Assignment statement
<i>PRINT</i>	127	Control statement
<i>PUT</i>	129	I/O statement
<i>READΔINTO</i>	130	I/O statement
<i>REF</i>	129	Called by <i>ASSIGNΔTO</i> , etc.
<i>REWIND</i>	126	Control statement
<i>SETPICTURE</i>	134	Called by <i>DECLARE</i>
<i>SHOWPICTURES</i>	126	Declarative statement
<i>SKIP</i>	135	Used in branching
<i>STACK</i>	129	Control statement
<i>STRETCH</i>	136	Enlarges symbol table
<i>SUB</i>	126	Assignment statement
<i>SUFFIX</i>	136	Deletes part of a vector
<i>VALUE</i>	134	Finds a qualifier
<i>WRITEΔFROM</i>	126	I/O statement
<i>Δ</i>	136	Tests "type" of an agrument

▽ SHOWPICTURES K;I;J;L;M

```
[1]  ' '
[2]  ('RECORD NO.  ' ;K)
[3]  →((K←K+1)>(ρIDENT)[1])/I←0
[4]  '*****
      NAME          PICTURE          POSITIONS          RANK'
[5]  →((ρIDENT)[2]<I←I+1)/0
[6]  →(' '=L←IDENT[K;I])/0
[7]  J←((0=ρJ)/0),J←K REF L
[8]  ' '
[9]  (IDENT[K;I],(6ρ' '),PICT[K;I;],2ρ' ';L/J;' TO ' ;[ /J;
      6ρ' ';(0<M)/M←RANK[K;I;])
[10] →5
      ▽
```

▽ REWIND N

```
[1]  INDEX[N]←1
[2]  ENDFILE[N]←0
      ▽
```

▽ R←VEC SUB LIST;K;L

```
[1]  →NAMEΔTYPE VEC
[2]  'ERROR'
[3]  →0
[4]  VEC←VALUE VEC
[5]  K←(÷6)×NUMΔSIGΔBITS L←PICT[VEC[2];VEC[1];]
[6]  R←RANK[VEC[2];VEC[1];]
[7]  →SKIP~'α'∈L
[8]  K←⌊K÷×/(R>0)/R
[9]  R←((R>0)/R)⊥-1+LIST
[10] R←VEC[1 2 ,2+(K×R)+⋮K]
      ▽
```

▽ N WRITEΔFROM K;L;M;O;P

```
[1]  →(0>TALLY[K←K+1])/0
[2]  →SKIP(ρDATASET)[1]≥INDEX[N]+1+TALLY[K]
[3]  DATASET←(1,1+TALLY[K])EXPANDK DATASET
[4]  DATASET[INDEX[N]+⋮TALLY[K];(6+N-1)+⋮6]←CORE[(RECL×K-1
      )+⋮TALLY[K];]
[5]  DATASET[INDEX[N];(6×N-1)+⋮6]←'9' FIXΔCODE TALLY[K]
[6]  END[N]←INDEX[N]←1+INDEX[N]+TALLY[K]
[7]  TALLY[K]←-⋮TALLY[K]
      ▽
```


▽ DECLARE K;I;N;J

```
[1]  N←0≠1|K
[2]  K←1+|K
[3]  →SKIP 0≥I←(RECL×K)-(ρCORE)[1]
[4]  →SKIP 7×0<×/ρIDENT
[5]  IDENT←(K,5)ρ' '
[6]  TABLE←(K,5,RECL)ρ0
[7]  PICT←(K, 5 15)ρ' '
[8]  RANK←(K, 5 3)ρ0
[9]  '  FOR EACH ITEM, TYPE:
      IDENTIFIER
      PICTURE',(K>1)/'
      INITIAL COLUMN NO.'
[10] SETPICTURE K
[11] →0
[12] 1 STRETCH K-(ρIDENT)[1]
[13] IDENT[NρK;]←' '
[14] SETPICTURE K
```

▽

▽ A MOVETO B;I;J;K;L;M

```
[1]  →(Λ/0<(ρ,I←NAMEΔTYPE A),ρ,J←NAMEΔTYPE B)/4
[2]  'UNDEFINED SYMBOLS'
[3]  →0
[4]  →SKIP I=5
[5]  A←VALUE A
[6]  →SKIP J=5
[7]  B←VALUE B
[8]  CORE[(RECL×B[2]-1)+I←2 SUFFIX B;]←(J,6)ρ(,CORE[(RECL×
A[2]-1)+2 SUFFIX A;]),(6×J←2+ρB)ρ0
[9]  TALLY[B[2]]←[ /TALLY[B[2]],I
```

▽

▽ PRINT N;I;LEN;SUB

```
[1]  SUB←(6×N-1)+16
[2]  I←1
[3]  LEN←21DATASET[I;SUB]
[4]  ALPHATOCHAR,DATASET[I+1LEN;SUB]
[5]  →(END[N]>I←I+LEN+1)/3
```

▽

▽ R←INITIAL N

```
[1]  R←N+0.5
```

▽

▽ CLOSE△FILES

[1] EMPTY△BUFFER

[2] BUF←10

▽

▽ R←CONTENTSOF VEC;I;J;K

[1] →NAME△TYPE VEC

[2] 'ERROR IN ''CONTENTSOF''.'

[3] →0

[4] VEC←VALUE VEC

[5] R←,CORE[(RECL×VEC[2]-1)+2 SUFFIX VEC;]

[6] R←PICT[VEC[2];VEC[1];]EXTERNAL R

▽

▽ GET NAMES;I;V;VAL;P1;J;K;L

[1] L←ITEM△IN+I←0

[2] NAMES←,NAMES

[3] INCA:→((ρNAMES)<I←I+1)/0

[4] →(0=ρBUF)/RD

[5] V←VALUE NAMES[I]

[6] →(∼v/J←v/[2]BUF)/RD

[7] P1←(-^/J)+(J^1φ∼J)11

[8] VAL←ALPHA△TO△CHAR,BUF[1P1;]

[9] BUF←BUF[P1+1(ρBUF)[1]-P1;]

[10] →SKIP 'α'εPICT[V[2];V[1];]

[11] VAL←10 NUMREP VAL

[12] VAL ASSIGN△TO V

[13] →INCA

[14] RD:→(∼ENDFILE[L]←INDEX[L]≥END[L])/RD+3

[15] 'END OF FILE CONDITION'

[16] →ρBUF←10

[17] VAL←21DATASET[INDEX[L];(6×L-1)+16]

[18] BUF←DATASET[INDEX[L]+1VAL;(6×L-1)+16]

[19] INDEX[L]←INDEX[L]+VAL+1

[20] →INCA+2

▽

▽ A1 MOVE△BY△NAME△TO A2;I;J;K;L

[1] →(A1=A2)/0

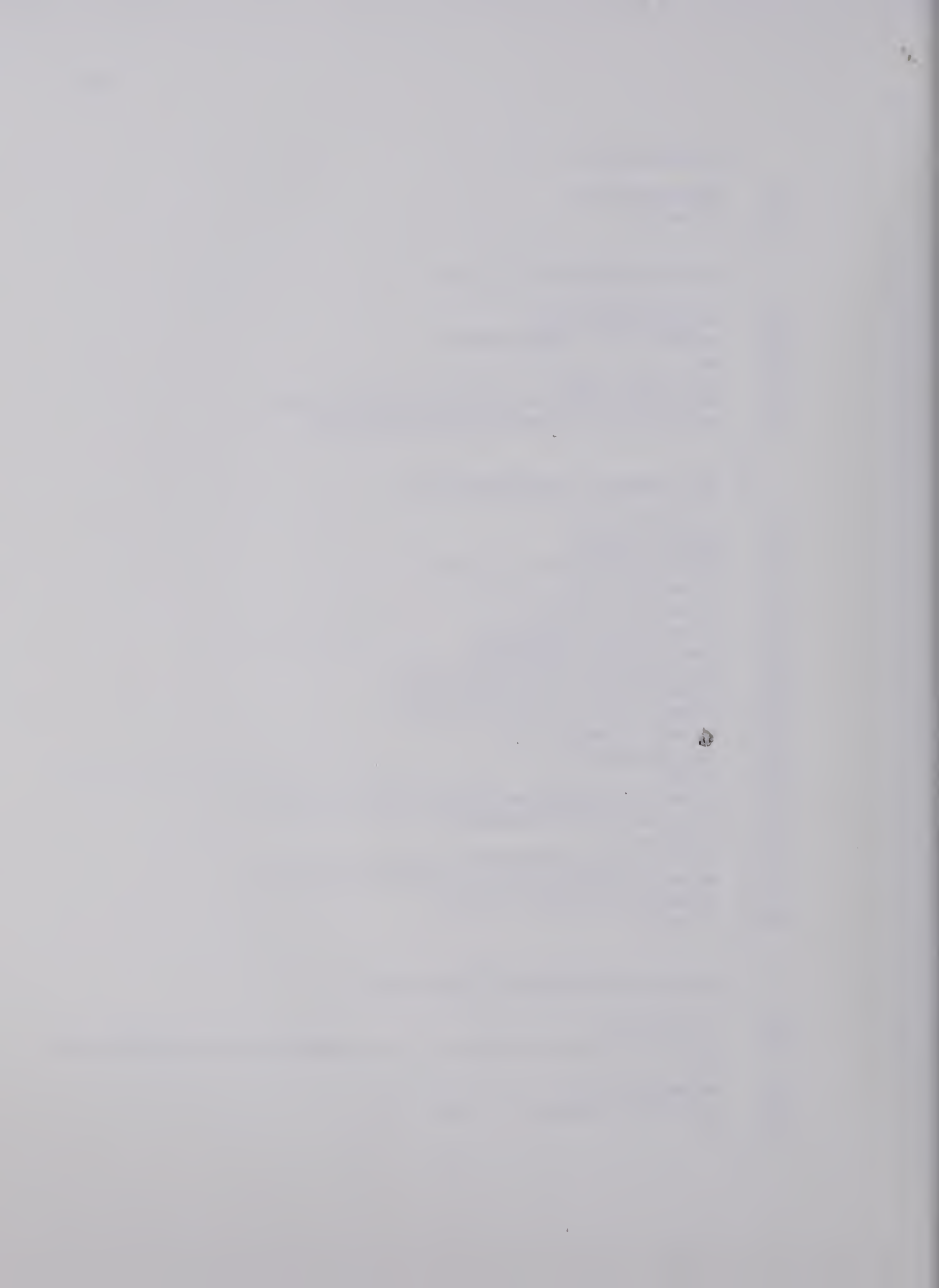
[2] J←1+ρI←,(v/[1]IDENT[1+A1;]◦.=IDENT[1+A2;])/IDENT[1+A2;]

[3] →(0≥J←J-1)/0

[4] (I[J]OF A1)MOVETO I[J]OF A2

[5] →3

▽



▽ N STACK STR; SUB; T; R; I; J

```
[1] SUB←(6×N-1)+16+I←0
[2] T←(((1+RECL)⌊ρ,STR)ρSTR), ' '
[3] DATASET[INDEX[N]; SUB]←(6ρ2)τρT
[4] →((ρDATASET)[1]≤INDEX[N]+ρT)/8
[5] →((ρT)<I+I+1)/10
[6] DATASET[INDEX[N]+I; SUB]←(6ρ2)τ-11+DATASEQ⌊T[I]
[7] →5
[8] DATASET← 1 10 EXPANDK DATASET
[9] →4
[10] END[N]←INDEX[N]←INDEX[N]+1+ρT
[11] →(0<ρSTR←,(ρ,T)SUFFIX STR)/2+I←0
```

▽

▽ R←NAME OF REC; I; J; K

```
[1] R←J, REC, (, TABLE[REC; J←IDENT[REC+1+REC; ]⌊NAME; ])/⌊RECL
```

▽

▽ R←K REF A

```
[1] K←K+1
[2] R←(, TABLE[K; IDENT[K; ]⌊A; ])/⌊RECL
```

▽

▽ PUT NAMES; I; VEC; F; PT; J; K; LTEMP

```
[1] NAMES←NAMES, ⌊I←0
[2] PT←OUTPTR
[3] ΔIND:→((ρNAMES)<I+I+1)/0
[4] VEC←VALUE NAMES[I]
[5] →SKIP 3×v/'Rα'∈F←PICT[VEC[2]; VEC[1]; ]
[6] F←('__', (EXΔFORMAT F), 'R__')EPORTCODE CONTENTSOF VEC
[7] TEMP←((J+⌊(ρF)÷6), 6)ρF
[8] →SKIP 2
[9] J←-2+ρVEC
[10] TEMP←CORE[2 SUFFIX VEC; ]
[11] →(RECL≤PT+J)/PUTPUT
[12] OUT[PT+⌊J; ]←TEMP
[13] OUTPTR←PT←PT+J
[14] →ΔIND
[15] PUTPUT:EMPTYΔBUFFER
[16] PT←0.
[17] →12
```

▽

▽ R←K GROUP LIST;I;J;L;M

```
[1] L←(∼LIST∈IDENT[K;])/LIST
[2] (0<ρL)/L,': UNDEFINED SYMBOLS IGNORED.'
[3] LIST←(LIST∈IDENT[K;])/LIST
[4] R←v/[1]TABLE[K;IDENT[K;]∖LIST;]
▽
```

▽ R←NN EXPANDK ARRAY;I;J

```
[1] I←NN[1]
[2] R←(((ρARRAY)[I]ρ1),NN[2]ρ0)∖[I]ARRAY
▽
```

▽ ARG ASSIGNΔTO VARB;I;J;K;L

```
[1] →NAMEΔTYPE VARB
[2] 'ERROR IN 'ASSIGNΔTO''.'
[3] →0
[4] VARB←VALUE VARB
[5] →(0≥J←2+ρVARB)/0
[6] K←(J,6)ρPICT[VARB[2];VARB[1];]INTERNAL ARG
[7] CORE[(RECL×VARB[2]-1)+J+2 SUFFIX VARB;]←K
[8] TALLY[VARB[2]]←[/TALLY[VARB[2]],J
▽
```

▽ BLANKΔRECORD K;I

```
[1] CORE[(RECL×K)+∖RECL;]←0
[2] TALLY[K+1]←RECL
▽
```

▽ CLEARΔPICTURES

```
[1] IDENT←TABLE←PICT←∖0
▽
```

▽ CLEARΔTAPES

```
[1] END←INDEX←3ρ1
[2] DATASET← 100 18 ρ0
[3] BUF←''
▽
```

▽ N READΔINTO K;TEMP;J;L

```
[1] →(ENDFILE[N]←INDEX[N]≥END[N])/0
[2] TEMP←2∖,DATASET[INDEX[N];L←(6×N-1)+∖6]
[3] CORE[(RECL×K)+∖TEMP;]←DATASET[INDEX[N]+∖TEMP;L]
[4] INDEX[N]←INDEX[N]+TEMP+1
[5] TALLY[K+1]←TEMP
▽
```


▽ R←FIELD ELEMENT A;N;VAL;T;U;V;J

```
[1] T←~'T'∈FIELD
[2] U←(FIELD←EXΔFORMAT(~FIELD∈' T')/FIELD)≠' _'
[3] FIELD←(FIELD∈'VYZ.,9X-+')/FIELD
[4] NUM:N←+/FIELD[ιJ←~1+(FIELD∈'XV.')ι1]∈'9'
[5] →(~'X'∈FIELD)/NUM+10
[6] N←N-[1+10⊙|A
[7] →(N<10*+/(VAL←FIELD[(FIELDι'X')+ι(ρFIELD)-FIELDι'X'])
   ∈'Z9Y')/NUM+5
[8] →(0<ρFIELD[(FIELD[ιJ]∈'ZY')/ιJ]←'9')/NUM
[9] →SKIP 2×N≠0
[10] R←'X 00'
[11] →SKIP 1
[12] R←'X',VAL ELEMENT-N
[13] →0,0/R←(FIELD[ι~1+FIELDι'X']ELEMENT A×10*N),R
[14] A←A×10*+/(FIELD[(FIELDι'V')+ι(ρFIELD)-(FIELD←FIELD,'V'
   )ι'V'])∈'9ZY'
[15] FIELD←(FIELD≠'V')/FIELD
[16] N←+/FIELD[ι~1+FIELDι'.' ]∈'9YZ'
[17] →((|A)<10*N)/NUMP9
[18] FIELD[((ρFIELD)ω3)/ιρFIELD]←'X+9'
[19] →NUM
[20] NUMP9:R←((+/FIELD∈'9ZY')ρ10)τ[(T×
   0.5)+1E~9+(|A)×10*( '.'∈FIELD)×(ρFIELD)-FIELDι'.'
[21] R←'0123456789'[R+1]
[22] R←R[ιN],('.'∈FIELD)/'.' ,R[N+ι(ρR)-N]
[23] R←((FIELD←U\FIELD)∈'ZY9.')\R
[24] U←((ρR)α~1+(R∈' 0')ι0)
[25] R[(FIELD∈'+-')/ιρFIELD]←(((A<0),A≥0),A≥0)^(((N+J),N←'
   +'∈FIELD),J←'- '∈FIELD))/'-+ '
[26] R[(FIELD∈',')/ιρFIELD]←', '
[27] R[((FIELD='Y')^R='0')/ιρR]←' '
[28] R[(U\FIELD∈'Z,')/ιρR]←' '
[29] →(v/R∈'0123456789')/0
[30] R←(ρR)ρ' '
▽
```

▽ R←BFLΔAPL BITS;I;X

```
[1] I←~7+ρBITS←,BITS
[2] R←(1-2×BITS[7])×+/(2*-ιI)×7 SUFFIX BITS
[3] R←R×8*1 BFXΔAPL BITS[ι6]
▽
```

▽ R←SW BFXΔAPL BITS;I;J;K

```
[1] BITS←((~SW)/0),BITS
[2] R←(1-2×BITS[1])×2ιBITS[1+ι~1+ρBITS]
▽
```


∇ R←F FIXΔCODE DEC;W;I

```
[1] I←v/'+-'∈F
[2] W←NUMΔSIGΔBITS F
[3] →SKIPv/(DEC≥0),'+-'∈F
[4] DEC←0
[5] →SKIP(|DEC)=2⊥R←((W-I)ρ2)⊥|DEC
[6] 'SIGNIFICANT BITS LOST'
[7] →(~I)/0
[8] R←(DEC<0),R
```

∇

∇ R←N CODEΔFRACT ARG;I;J;K

```
[1] →SKIP 1>|ARG
[2] →0=ρ□←'ERROR IN CODEΔFRACT'
[3] R←ARG<0
[4] ARG←8×1||ARG
[5] R←R, 2 2 2 ⊥|ARG
[6] →(N>ρR)/4
[7] R←NρR
```

∇

∇ R←F EXTERNAL BITS;I;J;K

```
[1] F←EXΔFORMAT F
[2] → 3 3 5 5 7[( 'αRX.'∈F)⊥1]
[3] R←ALPHΔTOΔCHAR BITS
[4] →0
[5] R←BFLΔAPL BITS
[6] →0
[7] R←(v/'+-'∈F)BFXΔAPL BITS
[8] →(~'V'∈F)/0
[9] R←R÷10*(ρF)-F⊥'V'
```

∇

∇ R←ALPHΔTOΔCHAR BITS;I;J

```
[1] R←⊥I←0
[2] BITS←((J←(ρBITS)÷6),6)ρBITS
[3] →(J<I←I+1)/0
[4] R←R,DATASEQ[1+2⊥BITS[I;]]
[5] →3
```

∇

▽ $R \leftarrow FMT \text{ INTERNAL ARG}; I$

```
[1]  FMT ← EXΔFORMAT FMT
[2]  → 3 5 7 7 9 [ ('αRX.' ∈ FMT) 1 1 ]
[3]  R ← FMT ALPHΔCODE ARG
[4]  → 0
[5]  R ← FMT EPORTCODE ARG
[6]  → 0
[7]  R ← FMT FLOATCODE ARG
[8]  → 0
[9]  R ← ((FMT ≠ 'V') / FMT) FIXΔCODE ARG
```

▽

▽ $R \leftarrow F \text{ ALPHΔCODE STR}; VEC; J$

```
[1]  R ← ''
[2]  J ← 1 + ρ VEC ← , -1 + DATASEQ 1 STR
[3]  → (0 ≥ J ← J - 1) / 6
[4]  R ← ((6 ρ 2) ↑ VEC[J]), R
[5]  → 3
[6]  R ← J ρ R, ((J ← 6 × ρ, F) - 6 × ρ, STR) ρ 0
```

▽

▽ $R \leftarrow FIELD \text{ EPORTCODE DEC}; I; J$

```
[1]  FIELD ← (~FIELD ∈ ' R') / FIELD
[2]  R ← ((ρ FIELD) ρ 'α') ALPHΔCODE FIELD ELEMENT DEC
```

▽

▽ $R \leftarrow FMT \text{ FLOATCODE ARG}; L; J; N; FR; I$

```
[1]  R ← (6 × L ← 4 + 7 × 5 <+ / FMT [ 1 J ← -1 + FMT 1 'X' ] ∈ '9Z') ρ 0
[2]  N ← 32 - (1 > (|ARG) × 8 × 32 - 163) 1 1
[3]  R [ 16 ] ← ' + 9 ' FIXΔCODE - N
[4]  R [ 6 + 16 × L - 1 ] ← (6 × L - 1) CODEΔFRACT ARG × 8 × N
```

▽

▽ $R \leftarrow NUMΔSIGΔBITS F; I; W; K$

```
[1]  R ← EXΔFORMAT F
[2]  → (0 < R ← 6 × + / F ∈ 'α') / 0
[3]  W ← + / (, F) [ 1 -1 + F 1 'X' ] ∈ 'Z9'
[4]  → SKIP 2 × v / 'X.' ∈ F
[5]  R ← 6 × ((10 1 W ρ 9) ≤ 2 × (-v / ' + - ' ∈ F) + 6 × 14) 1 1
[6]  → 0
[7]  R ← 24 + 42 × 5 < W
```

▽

▽ $R \leftarrow \text{VALUE NAME}; I; J; K; L$

```
[1]  R ← 1I + 0
[2]  → ((ρ IDENT)[1] < I + I + 1) / 0
[3]  J ← (K ← IDENT[I;] ∈ NAME) 11
[4]  → (~v / K) / 2
[5]  R ← J, I, I REF NAME
```

▽

▽ $\text{SETPICTURE } K; T; I; F; \text{SAVE}; L; FL; FX; J; P$

```
[1]  → (0 = ρ T ← □) / P ← 0
[2]  → SKIP ~' ° ' ∈ T
[3]  P ← □
[4]  I ← (, IDENT[K;] ∈ ' ', T ← 1ρ T) 11
[5]  2 STRETCH 5 × I > (ρ IDENT)[2]
[6]  IDENT[K; I] ← T
[7]  RANK[K; I;] ← 3ρ P, 0 0
[8]  P ← × / (P > 0) / P
[9]  T ← P × ρ EXΔFORMAT F ← □
[10] SAVE ← 1 + [ / 0, (v / [1] TABLE[K;;]) / 1 RECL
[11] → SKIP 4 × K = 1
[12] → SKIP 3 × 4 > Δ SAVE ← □
[13] TABLE[K; I;] ← K GROUP SAVE
[14] PICT[K; I;] ← ' [', ' 0123456789' [1+ 10 10 10 T + / TABLE[K; I;
    ], ' ] α ', 9ρ ' '
[15] → 1
[16] → SKIP 6 × ~' α ' ∈ F
[17] → SKIP 3 × K = 1
[18] SAVE ← ((ρ, SAVE) [ T ) ρ SAVE + 0, 1 T - 1
[19] TABLE[K; I;] ← (1 RECL) ∈ SAVE
[20] → SKIP 5
[21] SAVE ← 1 + SAVE + 1 P × ρ EXΔFORMAT F
[22] → SKIP 8
[23] → SKIP 4 × ' R ' ∈ F
[24] FX ← ( ÷ 6 ) × NUMΔSIGΔBITS F
[25] FL ← 4 + 7 × 6 ≤ + / L [ 1 1 + ( L ← EXΔFORMAT F ) 1 ' X ' ] ∈ ' Z 9 '
[26] L ← P × ( L, ~ L ← v / ' . X ' ∈ F ) / FL, FX
[27] → SKIP 1
[28] L ← P × + / ~ ( EXΔFORMAT F ) ∈ ' RTV '
[29] → SKIP L = ρ, SAVE
[30] SAVE ← ( 1 + 1 ρ SAVE ) + 1 L
[31] TABLE[K; I;] ← (1 RECL) ∈ SAVE
[32] PICT[K; I;] ← 15ρ F, 14ρ ' '
[33] → 1
```

▽

▽ $R \leftarrow B \text{ NUMREP } STRING; SEQ; FRACT; TEMP; I; SIGN$

```
[1]  →(0=ρR←STRING)/0
[2]  SEQ←'0123456789ABCDEF'[1B]
[3]  SIGN←(1 1 1)['-']1 (STRING←,(STRING≠' ')/STRING)[1]]
[4]  FRACT←(TEMP∈SEQ)/TEMP←(TEMP∈SEQ,'. ')/TEMP←STRING[1
      1+I←STRING1'X']
[5]  R←(SIGN×B1 1+SEQ1FRACT)×B×('.'∈TEMP)×(TEMP1'.' )-ρTEMP
[6]  →(I≥ρSTRING)/0
[7]  STRING←I SUFFIX STRING
[8]  R←R×B×10 NUMREP STRING
```

▽

▽ $EMPTY \Delta BUFFER; PT; M; L; I; J; K$

```
[1]  L←ITEMOUT
[2]  PT←OUTPTR
[3]  →((ρDATASET)[1]≤INDEX[L]+PT)/9
[4]  DATASET[INDEX[L];(6×L-1)+16]←(6ρ2)1PT
[5]  DATASET[INDEX[L]+1PT;(6×L-1)+16]←OUT[1PT;]
[6]  END[L]←INDEX[L]←INDEX[L]+PT+1
[7]  OUT←(RECL,6)ρOUTPTR←0
[8]  →0
[9]  DATASET← 1 10 EXPANDK DATASET
[10] →3
```

▽

▽ $R \leftarrow SKIP \ N; K$

```
[1]  R←' '
[2]  →((0=N)∨1≥ρK←,127)/0
[3]  R←N+K[1+11]+(-N<0)+N>0
```

▽

▽ $R \leftarrow EX \Delta FORMAT \ A; I; J; K; L$

```
[1]  R←,A
[2]  J←ρ,R
[3]  →(Λ/J<I←R1'[]')/0
[4]  L←101 1+'0123456789'1((Jα1+I[2])^~JαI[1])/R
[5]  R←R[1 1+I[1]],(LρR[1+I[2]]),(1+I[2])SUFFIX R
[6]  →2
```

▽

▽ $R \leftarrow \Delta \text{ ARG}$

```
[1] → (0 = ρ R ← ARG) / 0
[2] R ← 1
[3] → (Λ / ARG ∈ 1 0) / 0
[4] R ← 4
[5] → (V / ARG ∈ CHARSET) / 0
[6] R ← 2
[7] → (Λ / 0 = 1 | ARG) / 0
[8] R ← 3
```

▽

▽ $R \leftarrow N \text{ SUFFIX VECT}$

```
[1] R ← (¬(ρ VECT) α N) / VECT ← , VECT
```

▽

▽ $\text{LOAD} \Delta \text{ BUF}; L; \text{VAL}$

```
[1] L ← ITEM Δ IN
[2] → (¬ ENDFILE[L] ← INDEX[L] ≥ END[L]) / 4
[3] → ρ BUF ← 0 / 'END OF FILE'
[4] VAL ← 2 Δ DATASET[INDEX[L]; (6 × L - 1) + 16]
[5] BUF ← DATASET[INDEX[L] + 1 VAL; (6 × L - 1) + 16]
[6] INDEX[L] ← INDEX[L] + VAL + 1
```

▽

▽ $R \leftarrow \text{NAME} \Delta \text{ TYPE } X; I; J; K$

```
[1] I ← 2 ≤ ρ , X
[2] K ← 4 > Δ X
[3] J ← Λ / X ∈ IDENT
[4] R ← ((J ∧ ¬ I), I ∧ K) / 4 5
```

▽

▽ $\text{SUB STRETCH } N$

```
[1] → (N ≤ 0) / 0
[2] IDENT ← (SUB ← SUB, N) EXPANDK IDENT
[3] TABLE ← SUB EXPANDK TABLE
[4] PICT ← SUB EXPANDK PICT
```

▽

C.2.2 The Universal Format Language (UFL) System

<u>Function Name</u>	<u>Page</u>	<u>Remarks</u>
<i>CHARSET</i>	142	APL character vector
<i>CHECKΔFORM</i>	142	To enlarge <i>FORMM</i>
<i>CHECKΔTARGET</i>	141	To enlarge <i>TARGMAT</i>
<i>COMPL1</i>	139	"Compile" PL/1
<i>COMPΔFORT</i>	138	"Compile" Fortran
<i>DECIMAL</i>	140	Digits-to-number
<i>DECLCODE</i>	140	Number-to-digits
<i>DSF</i>	43	Initializes for subset of Fortran
<i>EXPANDK</i>	130	To enlarge any array
<i>FORTFORM</i>	118	Initializes for Fortran
<i>GO</i>	47	Called by <i>NEXTITEM</i>
<i>NEXTITEM</i>	47	Parsing function
<i>OUTITEM</i>	142	Executes output formats
<i>PLFORM</i>	122	Initializes for PL/1
<i>SEMANTICS</i>	141	Called by <i>NEXTITEM</i>
<i>SKIP</i>	135	Used in branching
<i>SUFFIX</i>	136	Deletes part of a vector
<i>TOP</i>	141	} To unstack one element
<i>TOPNUM</i>	141	
<i>TOPPAREN</i>	140	
<i>WRITEOUT</i>	141	Calls <i>OUTITEM</i>

▽ COMPΔFORT M;K;NSF;W;J;K

```

[1] →(M=0,(16),9+15)/(NSF+0),ONE,TWO,THR,FOUR,FIVE,SIX,
    TEN,ELVN,TWELVE,THIRT,FORT
[2] →0
[3] ONE:→ρ0/NUMSTACK←NUMSTACK,DECIMAL FORM[-1+J+1SYPTR-J←
    (,IDS)[ρ,IDS]]
[4] TEN:→ρ0/CURTYPE←'AIDFF'1FORM[SYPTR-1]
[5] THR:NSF←TOPNUM
[6] TWO:FORMM[;FNEXT]←((1+9×CURTYPE>1),1,CURTYPE=1),1,(W←
    TOPNUM),MNEXT
[7] →(CURTYPE=1)/0
[8] CHECKΔTARGET MNEXT+W
[9] TARGMAT[;MNEXT+1W]←0
[10] TARGMAT[1;MNEXT+1W]←CHARSET1'*'
[11] →SKIP 2×CURTYPE=2
[12] TARGMAT[3;(K←MNEXT)+W-NSF]←1
[13] →SKIP 1
[14] NSF←-1
[15] →SKIP 2×CURTYPE∈ 3 4
[16] TARGMAT[4;J←MNEXT+1W-NSF+2]←2
[17] TARGMAT[2;J]←1
[18] MNEXT←MNEXT+W
[19] →(CURTYPE∈ 2 5)/0
[20] J← 0 6 1 ,(NSFρ0), 4 0 0 0
[21] TARGMAT[3;K+1W]←((0[W-ρJ)ρ5),J
[22] TARGMAT[4;K+1W]←2×(1W)∈W-2,6+NSF
[23] →ρ0/TARGMAT[5;K+W-3]←CHARSET1'DE' [-2+CURTYPE]
[24] SIX:FORMM[4;FNEXT]←TOPNUM
[25] FOUR:FNEXT←FNEXT+1
[26] CHECKΔFORM
[27] →0
[28] TWELVE:→ρ0/PARENSTACK←PARENSTACK,FNEXT
[29] THIRT:FORMM[;FNEXT]← 0 0 0 ,TOPNUM,0,TOPPAREN
[30] →FOUR
[31] FORT:→ρ0/TOPPAREN
[32] FIVE:→SKIP 2×'X'≠FORM[SYPTR-1]
[33] FORMM[;FNEXT]← 0 0 0 0 ,TOPNUM,0
[34] →FOUR
[35] W←TOPNUM+K+0
[36] NSF←-1+SYPTR+1W
[37] SYPTR←SYPTR+W+K
[38] CHECKΔTARGET MNEXT+W
[39] TARGMAT[;MNEXT+1W]←(5,W)ρ(CHARSET1FORM[NSF]),(
    5×W)ρ0
[40] FORMM[;FNEXT]← 1 0 1 0 ,W,MNEXT
[41] MNEXT←MNEXT+W
[42] →FOUR
[43] ELVN:W←(SYPTR SUFFIX FORM)1''''
[44] K←1
[45] →4+FIVE

```

▽

∇ COMPL1 N;I;W;D;J;K;MAT;A;B;C

```

[1]  →(N= 2 8 9 10 ,12+17)/PL2,PL8,PL9,PL10,PL13,PL14,PL15
      ,PL16,PL17,PL18,PL19
[2]  →0
[3]  PL2:NUMSTACK←NUMSTACK,DECIMAL FORM[-1+I+1SYPTR-I←(,
      IDS)[ρ,IDS]]
[4]  →0
[5]  PL8:VEC←VEC,TOPNUM
[6]  →(CURTYPE≤3)/ABX
[7]  MAT←(5,W←VEC[1])ρ0
[8]  →SKIP 3×CURTYPE≠4
[9]  MAT[2 4 ;]← 1 2 0.×(Φ1W)>1+D+0<D←(VEC,0)[2]
[10] MAT[3;W-D]←D>0
[11] →DOOO
[12] →SKIP 3×2≤ρVEC
[13] 'WRONG SPECIFICATION: ',FORM
[14] ((19+SYPTR)ρ' '),'+ '
[15] VEC←VEC,0
[16] VEC←3ρVEC,(1+VEC[2]),0
[17] MAT[3;I←((Φ1W)∈4,5+VEC[2])/1W]← 1 4
[18] MAT[5;I[2]]←CHARSET1'E'
[19] MAT[4;(W-2),I←1[W-VEC[3]+5]← 3 2
[20] MAT[3;1I-1]←5
[21] →DOOO
[22] ABX:→SKIP CURTYPE≠3
[23] →1ρFORMM[;FNEXT]← 0 0 0 0 ,VEC[1],0
[24] FORMM[;FNEXT]←CURTYPE, 0 0 1 ,VEC[1],0
[25] PL9:VEC←10
[26] CURTYPE←'ABXFE'1FORM[SYPTR-1]
[27] →COMCNT←0
[28] PL19:→SKIP 2× 0 0 0 1 2[CURTYPE]≥COMCNT←COMCNT+1
[29] 'ILLEGAL COMMA: ',FORM
[30] ((13+SYPTR)ρ' '),'+ '
[31] VEC←VEC,TOPNUM
[32] PL10:→(,FORMM[;FNEXT]←CURTYPE, 0 1 1 0 0)[2]
[33] PL13:STR←FORM[-1+I+1SYPTR-I←IDS[ρ,IDS]]
[34] →SKIP 2×((∨/'*Z'∈STR)×[/(STR∈'*Z')/1ρSTR)<W←STR1'9'
[35] 'ILLEGAL SEQUENCE: ',STR
[36] ((17+W+(W SUFFIX STR∈'*Z')11)ρ' '),'+ '
[37] K←('V'∈STR)×((~STR∈'P''')/STR)1'V'
[38] W←ρSTR←(~STR∈'VP''')/STR
[39] →(∨/'AX'∈STR)/PICTU
[40] MAT←(5,W)ρ0
[41] MAT[2;]←Wα-1+STR1'9'
[42] MAT[3;]←(7×STR='.')+(2×STR=',')+(3×STR='/' )+
      5×STR='B'
[43] MAT[4;]←(STR='+')+(2×STR='-')+3×STR='S'
[44] MAT[5;]←(STR='*')×CHARSET1'*'
[45] →SKIP 0=K

```



```

[45] →SKIP 0=K
[46] MAT[3;K]← $\overline{1+2 \times 7}$ =MAT[3;K]
[47] DOOO:CHECKΔTARGET MNEXT+W
[48] TARGMAT[;MNEXT+ $\imath$ W]←MAT
[49] FORMM[;FNEXT]← 10 0 0 1 ,W,MNEXT
[50] MNEXT←MNEXT+W
[51] →0
[52] PICTU:→(,FORMM[;FNEXT]← 1 0 1 1 ,W,0)[2]
[53] PL14:→ρ0/TOPPAREN
[54] PL16:FORMM[4;FNEXT]←TOPNUM
[55] PL15:FNEXT←FNEXT+1
[56] CHECKΔFORM
[57] →0
[58] PL17:FORMM[;FNEXT]← 0 0 0 ,TOPNUM,0,TOPPAREN
[59] →PL15
[60] PL18:PARENSTACK←PARENSTACK,FNEXT

```

▽

▽ R←N DECLCODE ARG;I;J;K;L;M;B

```

[1] K←N[2]
[2] R←(N←1ρN)ρ'0'
[3] →(0=ARG←|ARG)/0
[4] →SKIP 7×ARG> $\overline{1+2 \times 32}$ 
[5] ARG←ARG+0.5×10*-K
[6] R←((0[N-K]ρ10)τ|ARG
[7] →SKIP 2×K=0
[8] R←R,|ARG←10×1|ARG
[9] →(N>ρR)/8
[10] R←'0123456789'[1+R]
[11] →0
[12] R←Nρ'*'
[13] →(v/(ARG≥10*N),N>16)/0
[14] R←((9,K[9],J)DECLCODE(10*9)|ARG
[15] ARG←|ARG÷10*9
[16] R←(((N-9),0[K-9)DECLCODE ARG),R

```

▽

▽ R←DECIMAL STR;I;J;K;L

```

[1] STR←(STRε'0123456789')/STR
[2] R←10 $\perp$  $\overline{1+}$ '0123456789' $\imath$ STR

```

▽

▽ R←TOPPAREN

```

[1] R←(,PARENSTACK)[ρ,PARENSTACK]
[2] PARENSTACK←PARENSTACK[ $\imath$  $\overline{1+}$ ρ,PARENSTACK]

```

▽

▽ STR←WRITEOUT LIST;I;REPTAB;REP;ST;J;K;X;Y;Z

```
[1]  STR←1I←0
[2]  LOOP:→SKIP 1<I←1+(FNEXT-1)|I
[3]  REPTAB← 7 2 ρ0
[4]  →SKIP 3×FORMM[5;I]≠REP←0
[5]  →(FORMM[4;I]≤K←REPTAB[J←(REPTAB[;1]∈0,I)1;2]+1)/LOOP
[6]  REPTAB[J;]←I,K
[7]  I←FORMM[6;I]
[8]  →SKIP 2×FORMM[1;I]≥1
[9]  STR←STR,FORMM[5;I]ρ' '
[10] →LOOP
[11] →SKIP 2×0≠FORMM[4;I]
[12] STR←STR,CHARSET[TARGMAT[1;FORMM[6;I]+1FORMM[
    5;I]]]
[13] →LOOP
[14] STR←STR,FORMM[;I]OUTITEM 1ρLIST
[15] →SKIP 2×0=ρLIST←1 SUFFIX LIST
[16] →SKIP 1×FORMM[4;I]>REP←REP+1
[17] →LOOP
```

▽

▽ CHECKΔTARGET N;I;J;K;L

```
[1]  →(0>K←N-(ρTARGMAT)[2])/0
[2]  TARGMAT←(2,K)EXPANDK TARGMAT
```

▽

▽ SEMANTICS N

```
[1]  →2×LANGCODE
[2]  COMPΔFORT N
[3]  →0
[4]  COMPL1 N
```

▽

▽ R←TOPNUM

```
[1]  R←(,NUMSTACK)[ρ,NUMSTACK]
[2]  NUMSTACK←(~(ρ,NUMSTACK)ω1)/NUMSTACK
```

▽

▽ R←TOP;K

```
[1]  R←(,ROWPT)[K←ρ,ROWPT]
[2]  ROWPT←(~Kω1)/ROWPT
```

▽

▽ *R←M OUTITEM A;Y;L;B;D;N;Z;J;SAVE;EXP;U;K;S*

```

[1]  Y←TARGMAT[;M[6]+1(M←,M)[5]]
[2]  R←(L←M[5])ρ' '
[3]  →SKIP 4×1<B←M[1]
[4]  D←+/Y[1;]=0
[5]  N←((~M[3])×0[D-ρ,A])ρ' '
[6]  R[(Y[1;]=0)/1L]←DρN,A,R
[7]  LAST:→ρ0/R[U/1L]←CHARSET[(U←Y[1;]>0)/Y[1;]]
[8]  Z←(J←1+Y[3;]14)>1+(Y[4;]>0)11
[9]  Z←Z∧~((A<0)∧1∈Y[4;])∨(A>0)∧2∈Y[4;]
[10] Z←Z∧S←(4∈Y[3;])∨(|A|<10*+/(~Z),0≥Y[3;]11+(Y[
    3;]∈11)11
[11] SAVE←+/D<0≥Y[3;Z+1L-Z]
[12] →SKIP 9×~4∈Y[3;]
[13] N←((Y[3;]∈16)11)-1+(Y[4;]>0)11
[14] N←N-[1+10⊕|A|
[15] EXP←('X',CHARSET)[1+Y[5;J+1]]
[16] S←(((N≤0)∧Y[4;J+2]∈13),(N>0)∧Y[4;J+2]∈23)/'+-'
[17] EXP←EXP,(1ρS,' '),20DECLCODE|N
[18] R←((10,M[2],01,J,M[6])OUTITEM A×B×N),EXP
[19] →0
[20] R[D/Z+1L-Z]←SAVEρ((~M[2])+SAVE,L-[/(Y[3;]11),
    1+Y[3;]11)DECLCODE A
[21] R[(Y[3;]∈17)/1L]←'.'
[22] R[(Y[3;]=2)/1L]←','
[23] R[(Y[3;]=3)/1L]←'/'
[24] R[(Y[3;]=5)/1L]←' '
[25] U←Lα1+(R∈'123456789')11
[26] R[(U∧Y[2;]∨Y[4;]∈13)/1L]←' '
[27] R[(Y[3;]=6)/1L]←'0'
[28] →SKIP 4×Z=0
[29] →SKIP~∨/(A<0)∧D←U∧Y[4;]∈23
[30] R[|D/1L]←'-'
[31] →SKIP~∨/(A>0)∧D←U∧Y[4;]∈13
[32] R[|D/1L]←'+'
[33] U←Lα1+(R∈'123456789')11
[34] →SKIP 0=+/U←U∧Y[5;]>0
[35] R[U]←CHARSET[Y[5;U←U/1L]]
[36] →(~S)/LAST

```

▽

▽ *CHECKΔFORM*

```

[1]  →(FNEXT≤(ρFORMM)[2])/0
[2]  FORMM←25EXPANDK FORMM

```

▽

CHARSET

```

""<=>≠∨∧÷?ω∈ρ~↑↓10*→α[[_∇Δ°'□()c>null|;:\
1234567890+×QWERTYUIOP←ASDEFGHJKL[]ZXCVBNM,./

```


B29914